The Dissertation Committee for John D. Erickson

certifies that this is the approved version of the following dissertation:

Generalization, Lemma Generation, and Induction in ACL2

Committee:

J Strother Moore II, Supervisor

Clark Barrett

Matt Kaufmann

Vladimir Lifschitz

Jayadev Misra

Generalization, Lemma Generation, and Induction in ACL2

by

John D. Erickson, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2008

To Nikki

Acknowledgments

I would like to thank J Moore for his encouragement and support. His depth of knowledge about theorem proving never ceased to amaze me. He was always able to offer me some kind of wisdom, no matter how lost I was. I would like to thank Matt Kaufmann for relentlessly helping me to figure out how to solve all of the problems I had with ACL2 along the way. I can't imagine how anyone in the ACL2 community could survive without him. I would like to thank Warren Hunt for keeping funding flowing into our group, and teaching me a little about how the real world works. I would like to thank J Moore, Clark Barrett, Matt Kaufmann, Vladimir Lifschitz, and Jayadev Misra for serving on my committee. I would like to thank the ACL2 group at UT, and all the people on the 20th floor for the good times and stimulating conversations. This includes Bob Boyer, Matt Kaufmann, Warren Hunt, Hanbing Liu, J Moore, David Rager, Erik Reeber, Robert Krug, Sandip Ray, Jared Davis, Sol Swords, Chad Wellington, Bill Young, Qiang Zhang, and anyone else I may have forgotten. I would like to thank the greater ACL2 community for the great help on the acl2-help mailing list and the always fun ACL2 Workshop. I would like to thank Kata Carbone and Carol Hyink for all the meetings they set up for me. I would like to thank Gloria Ramirez and Katherine Utz for helping with all the various administrative tasks I've needed to do over the years. I actually enjoyed some parts of the UT bureaucracy because of them. This material is based upon work supported by DARPA and the National Science Foundation under Grant No.

CNS-0429591 and also National Science Foundation Grant No. EIA-0303609. I would like to thank all of the other CS grad students for the fun times, especially the time playing soccer, basketball, and foosball. Finally, I would like to thank my family, my friends, and my wife Nikki for sticking with me through all these years and helping me to keep things in perspective.

John D. Erickson

The University of Texas at Austin

May 2008

Generalization, Lemma Generation, and Induction in ACL2

Publication No. _____

John D. Erickson, Ph.D.
The University of Texas at Austin, 2008

Supervisor: J Strother Moore II

Formal verification is becoming a critical tool for designing software and hardware today. Rising complexity, along with software's pervasiveness in the global economy have meant that errors are becoming more difficult to find and more costly to fix. Among the formal verification tools available today, theorem provers offer the ability to do the most complete verification of the most complex systems. However, theorem proving requires expert guidance and typically is too costly to be economical for all but the most mission critical systems.

Three major challenges to using a theorem prover are: finding generalizations, choosing the right induction scheme, and generating lemmas. In this disser-

tation we study all three of these in the context of the ACL2 theorem prover.

Contents

Chapter 1

Introduction

Software pervades modern life. Cell phones, computers, and iPods have all become a part of everyday life and are brought to life by millions of lines of code. Today, software can even be found in many less obvious places, such as cars, toasters, airplanes, and medical equipment.

As we rely more on software, it becomes more important that it functions properly. However, today's software is more complex than ever before, and is increasingly becoming more complex. These factors combine to produce more hard to find bugs in important software than ever before. Standard techniques for finding bugs are no longer able to guarantee the level of assurance necessary for many tasks. Formal verification, which uses mathematical analysis to verify software, can help ameliorate this problem.

There are two main contribution of this dissertation. The first is a novel technique for generating proofs from proofs of finite cases. This technique works by finding patterns in proofs of finite cases of the goal theorem and then using these patterns to generate inductive proofs for the original theorem. We describe an

implementation of this algorithm that works for an interesting subset of the ACL2 logic and describe how we think the technique may be further extended. The second main contribution of this dissertation is to show how backtracking may be added to the ACL2 theorem prover. We describe an implementation that uses backtracking for several purposes. First, we can use it to find substitutions for non-induction variables during induction. Second, it can be used to speculate lemmas. Finally, it can be used to attempt multiple generalizations.

1.1 Formal Verification

There are many types of formal verification. They range from fully automatic techniques to techniques requiring much skill and effort. Some techniques include model checking, equivalence checking and symbolic trajectory evaluation. The key difference between formal verification and other types of verification is that formal verification uses mathematical algorithms to achieve 100% coverage. These formal models rigorously define the behavior of a given piece of software using symbolic logic. With such a model, it is possible to reason about a program symbolically, rather than just being able to execute it for some given inputs. This enables tools to draw conclusions about the program's behavior on classes of inputs rather than merely on one input at a time. Formal tools can even be used to draw conclusions about programs and algorithms that can visit infinitely many states, which could never all be tested.

In this dissertation, we focus on one particular type of formal verification called automated theorem proving. Theorem proving differs from other types of formal verification in several ways. The first and most important difference is the logic used for building and reasoning about models in theorem proving. Theorem

proving typically uses a powerful logic. This has two main effects. First, it allows the user to build more sophisticated models and check more properties of those models. Second, it makes it more difficult to automate the checking of properties. These two factors together mean that theorem proving is typically only used on projects where the cost of failure is quite large and it is worth the extra effort to check as much as possible.

There are many different provers available. Our work is focused on techniques for interactive inductive theorem proving. Some of the most popular theorem provers of this type include HOL4, PVS, Coq, and ACL2. Although they differ in the logics used and the heuristics guiding them, many of the principles used are similar. For example, induction and rewriting are two common reasoning techniques found in almost every theorem prover. For the purposes of this dissertation, we will be using the ACL2 theorem prover. ACL2 is descended from the line of Boyer-Moore theorem provers. One of its main design goals was to be an industrial strength theorem prover. This means that ACL2 is able to handle very large designs. Another key feature of ACL2 is that it is able to execute programs written in its logic as compiled LISP programs. This can offer several orders of magnitude improvements in efficiency of execution over other provers and is a critical feature for industrial applications because it can be used to check formal models against other models using large test suites.

1.2 Introduction to Using ACL2

ACL2 is both a programming language and a logic. It uses a purely functional subset of LISP for its programming language. A functional subset was chosen in part because it is relatively easy to reason about. In particular, the fact that there

are no procedures with side effects makes it easier to represent the effects of a function call symbolically. LISP syntax is very simple; although many new users find it difficult to read, its simplicity does have some merits when trying to write interpreters or other functions that operate on nested expressions. One thing to note is that function calls are written with the first parenthesis before the function name, rather than after it, and that there are no commas between function arguments. Thus foo(x, y) is written (foo x y).

A typical user interfaces with ACL2 through an interactive read, eval, print loop. When a user enters an expression into the loop, ACL2 will evaluate the expression and print the result. Some datatypes, like numbers, can be written the same before and after evaluation. Others, however, such as symbols and lists, need to be quoted before they are evaluated, otherwise they will be confused with variables or function calls.

The basic datatypes in ACL2 are numbers, strings, characters, and symbols. For purposes of this dissertation, we will only be concerned with natural numbers, which are written as one might expect, and symbols, which are typically preceded with a single-quote before evaluation, and are just printed out without any modifications afterwards. For example, the quoted symbol 'FOO evaluates to the symbol FOO.

Pairs are constructed using the function CONS. CONS takes two items and puts them together into a pair. Pairs are printed as two items separated by a dot inside a pair of parenthesis, e.g. (1 . 2) is the pair containing the numbers 1 and 2, respectively. Pairs can be nested inside one another by nested calls of CONS. E.g., (CONS 1 (CONS 2 3)) evaluates to (1 . (2 . 3)). By nesting pairs, we can create lists and trees. Lists are represented by right-associated nests of pairs "terminated"

by the symbol NIL. E.g., (1 . (2 . (3 . NIL))) is the list containing 1, 2, and 3. When a list follows this form we will usually write it using the simpler form (1 2 3). The macro LIST will create a nest of CONSes to form such a list, e.g. (LIST 1 2 3) evaluates to (1 2 3). The functions car and cdr can be used to access the elements of a pair. For example, (CAR (CONS 1 2)) is 1 and (CDR (CONS 1 2)) is 2. A common abbreviation used for (CAR (CDR (CDR x))) is (caddr x). This same style of abbreviation is used for any sequence of nested calls to car and cdr up to length 4. We can use singlequote to easily input large constants. For example, '(1 2 3) evaluates to (1 2 3), and '(FOO A B) evaluates to (FOO A B).

Functions are introduced into ACL2 with the DEFUN form. For example, the function ADD1 can be defined as:

```
(defun add1 (x) (+ 1 x))
```

Functions that in traditional programming languages might be written using FOR or WHILE loops are typically written using recursion in ACL2. For example, to find the number of elements in a list, we define the function LEN as such:

```
(defun len (x)
  (if (endp x)
      0
    (+ 1 (len (cdr x)))))
```

Note that the function ENDP tests a list to see if it is empty.

We can reason about the definitions we introduce by proving theorems about them. The DEFTHM form is used to introduce theorems. When a defthm is submitted, ACL2 will attempt to prove the given conjecture. If successful, it will add the theorem to ACL2's database of known facts. If the proof fails, ACL2 will

output a trace explaining how ACL2 attempted to prove the theorem and why it failed.

Below is an example of a successful proof in ACL2. It is a theorem about the associativity of the function ap, which is defined as:

```
(defun ap (x y)
  (if (endp x)
      y
    (cons (car x) (ap (cdr x) y))))
```

This function is used to concatenate two lists. For example, evaluating (ap '(1 2 3) '(4 5)) yields (1 2 3 4 5). We invoke the prover by submitting a defthm event:

```
(defthm assoc-ap
  (equal (ap (ap x y) z)
         (ap x (ap y z))))
```

In the next few pages, we show a typical ACL2 proof script (presented in typewriter font). The ACL2 prompt `ACL2 !>` indicates that ACL2 is waiting for some input. In this example, the user first enters the definition for the function *ap*. ACL2 checks this function to ensure that it terminates on all inputs, and then stores the definition. Next, the user enters the assoc-ap theorem. Here ACL2 outputs a great deal of text about how it is attempting to prove the theorem. It concludes successfully by storing the theorem. Although we do not expect the reader to understand all the details of this output, we thought it might be instructive to show how a user typically interacts with ACL2. After the end of this script, we go into some detail about some of the techniques ACL2 is using during the proof.

```
ACL2 !> (defun ap (x y)
```

```
          (if (endp x) y (cons (car x) (ap (cdr x) y))))
```

The admission of AP is trivial, using the relation O< (which is known to be well-founded on the domain recognized by O-P) and the measure (ACL2-COUNT X). We observe that the type of AP is described by the theorem (OR (CONSP (AP X Y)) (EQUAL (AP X Y) Y)). We used primitive type reasoning.

Summary

Form: (DEFUN AP ...)

Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))

Warnings: None

Time: 0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)

AP

```
ACL2 !>(defthm assoc-ap (equal (ap (ap x y) z) (ap x (ap y z))))
```

Name the formula above *1.

Perhaps we can prove *1 by induction. Three induction schemes are suggested by this conjecture. Subsumption reduces that number to two. However, one of these is flawed and so we are left with one viable candidate.

We will induct according to a scheme suggested by (AP X Y). This suggestion was produced using the :induction rule AP. If we let

```
(:P X Y Z) denote *1 above then the induction scheme we'll use is
(AND (IMPLIES (AND (NOT (ENDP X)) (:P (CDR X) Y Z))
              (:P X Y Z))
     (IMPLIES (ENDP X) (:P X Y Z))).
This induction is justified by the same argument used to admit AP.
When applied to the goal at hand the above induction scheme produces
the following two nontautological subgoals.

Subgoal *1/2
(IMPLIES (AND (NOT (ENDP X))
              (EQUAL (AP (AP (CDR X) Y) Z)
                     (AP (CDR X) (AP Y Z))))
         (EQUAL (AP (AP X Y) Z)
                (AP X (AP Y Z)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/2'
(IMPLIES (AND (CONSP X)
              (EQUAL (AP (AP (CDR X) Y) Z)
                     (AP (CDR X) (AP Y Z))))
         (EQUAL (AP (AP X Y) Z)
                (AP X (AP Y Z)))).

But simplification reduces this to T, using the :definition AP,
```

```
primitive type reasoning and the :rewrite rules CAR-CONS and
CDR-CONS.

Subgoal *1/1
(IMPLIES (ENDP X)
         (EQUAL (AP (AP X Y) Z)
                (AP X (AP Y Z)))).

By the simple :definition ENDP we reduce the conjecture to

Subgoal *1/1'
(IMPLIES (NOT (CONSP X))
         (EQUAL (AP (AP X Y) Z)
                (AP X (AP Y Z)))).

But simplification reduces this to T, using the :definition AP
and primitive type reasoning.

That completes the proof of *1.

Q.E.D.

Summary
Form:  ( DEFTHM ASSOC-AP ...)
Rules: ((:DEFINITION AP)
```

```
        (:DEFINITION ENDP)
        (:DEFINITION NOT)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION AP)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS))
Warnings:  None
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 ASSOC-AP
ACL2 !>
```

As with most theorems involving recursive functions, ACL2 will choose to induct to prove this theorem. The induction scheme is chosen heuristically based on the recursive functions present in the conjecture. In this case, it has chosen an induction on x by cdr. This induction involves two cases. First, the induction step must show that if we assume the theorem holds when we substitute (cdr x) for x, then we can prove the theorem holds for x. The base case requires that we show the theorem holds when (endp x) is true, in other words, when the list x is empty.

All induction in ACL2 must be justified by some measure decreasing for all induction steps. This ensures that the induction is well-founded. ACL2 allows measures over ϵ_0-ordinals, however, for the purposes of this dissertation, induction over naturals should be sufficient.

Subgoal *1/2 results from the induction step. It has a hypothesis that contains the original theorem with (cdr x) substituted for x and an additional hypothesis (not (endp x)). ACL2 simplifies this subgoal slightly in *1/2', replacing (not (endp x)) with (consp x). The remaining work done to complete this subgoal is not shown.

ACL2 does not show all of the steps it performs because they often would present far too much detail. In this case, we can imagine how ACL2 completes this theorem as it hints in the commentary to *1/2'. Using the definition of ap, under the assumption that x is a consp, it can simplify the theorem to

```
(IMPLIES (AND (CONSP X)
              (EQUAL (AP (AP (CDR X) Y) Z)
                     (AP (CDR X) (AP Y Z))))
         (EQUAL (CONS (CAR X) (AP (AP (CDR X) Y) Z))
                (CONS (CAR X) (AP (CDR X) (AP Y Z)))))
```

At this point, applying the induction hypothesis will complete the theorem.

For the base case, ACL2 uses a similar line of reasoning under the assumption that x is endp.

When the theorem completes, it prints out a summary of the lemmas used.

Chapter 2

Generalizing Finite Cases

In this chapter, we present a technique for generating inductive proofs by generalizing proof instances. This results in an automatic proof technique that can handle many theorems requiring generalization or lemma generation. We have implemented this technique using the ACL2 [16] theorem prover and have tested it on a variety of theorems. Our results show that this technique can automatically prove many theorems that ACL2 cannot.

An overview of our technique is as follows. First, we generate instances of the theorem we want to prove. For purposes of this paper, we restrict ourselves to theorems about naturals and linear lists, although we believe these techniques can be extended to handle other datatypes. These instances can easily be proved using simple rewriting techniques. Once we have generated proofs for a few instances, we apply a pattern recognition algorithm to the proof of each instance. This algorithm finds patterns in the proofs and replaces them with recursive functions that generalize the proof patterns it finds. The process iterates until a single recursive function is found that describes the general pattern of the proofs. This general

pattern is then used to generate a series of definitions and lemmas that give a first order inductive proof of the original theorem. Our implementation uses ACL2 to automatically check this proof. Since the proofs we generate are checked by ACL2, there is no need to trust that our method is correct.

Overall, our technique is similar to Pearson's [18]. However, we use a different representation for our proofs which allows us to recognize more complex patterns. This allows us to prove theorems that would otherwise require additional lemmas be given by the user.

One caveat about this approach is that pattern recognition is second nature to people. People intuitively pick up on patterns without much effort. In this work, we're trying not only to describe patterns formally with ACL2 formulas, but also describe algorithms for finding patterns. Since we are unable to leverage human intuition for this task, it can be quite messy. One interesting possible future direction for this work may be to consider a tool that is interactive and allows the user to do some of the work of detecting patterns.

We begin this chapter with a series on examples that gradually add complexity. We use these examples to present different aspects of our system. In later sections, we present a more complete description of our technique. More specifically, this chapter is structured as follows. Section 2.1 gives a sense of how we discover and take advantage of patterns in proofs. Section 2.2 formalizes the inductions used in Section 2.1. Such inductions also are used in the final step of our ultimate algorithm presented in Sections 2.10 and beyond. 2.3 through 2.8 give an introduction to more complex pattern discovery mechanisms, together with their application to simplified versions of our actual algorithm. Section 2.9 presents an example worked according to our actual algorithm. Sections 2.10 through 2.12 present our actual

algorithm. Section 2.13 presents results. Section 2.14 concludes. Appendix A describes attempts to generalize our pattern discovery to tree, which however are not part of our actual algorithm at this point. Accompanying this document is also an archive that includes code and runnable examples, available at
`http://www.cs.utexas.edu/~jderick/thesis-code.tgz`.

2.1 Motivating Example

Consider the theorem $ones(x) = ones1(x, nil)$, where the definitions for $ones$ and $ones1$ are given below. [1] Here we write the function cons using the infix symbol '::'.

$$ones(nil) = nil$$
$$ones(h :: t) = 1 :: ones(t)$$

$$ones1(nil, a) = a$$
$$ones1(h :: t, a) = ones1(t, 1 :: a)$$

In the theorem above, the second argument in the call to $ones1$ is the constant nil, while the definition for $ones1$ modifies its second argument when it recurs. This problem is typically referred to as a blocked accumulator, and makes the theorem impossible to prove directly by induction. In order to solve this problem, the theorem must be generalized. A typical generalization is $append(ones(x), a) = ones1(x, a)$. This generalization can be proved by induction. Notice that the generalization contains the function $append$, which is not used anywhere in the original theorem. Factors like these can make discovering generalizations automatically difficult.

[1] A very similar example could have been presented for the more commonly used functions len and $len1$. However, since the length of the list x is involved in the general framework presented here we use these functions instead to keep the two uses from being confused.

Our technique avoids this problem by finding a generalization based on the patterns found in a proof for a specific case of this theorem. Consider the case where x is a list of two elements. We can write a proof for this case as follows. Later we will discuss the details of how to choose a representative case and how to generate proofs for such a case, but for now assume we generate the following proof:

$$ones(x) = ones1(x, nil)$$
$$1 :: ones(cdr(x)) = ones1(cdr(x), 1 :: nil)$$
$$1 :: 1 :: ones(cdr(cdr(x))) = ones1(cdr(cdr(x)), 1 :: 1 :: nil)$$
$$1 :: 1 :: nil = 1 :: 1 :: nil$$

We will see how this proof suggests the alternative generalization:

$$nones(n, ones(nthcdr(n, x))) = ones1(nthcdr(n, x), nones(n, nil))$$

where *nthcdr* and *nones* are defined as:

$$nthcdr(0, x) = x$$
$$nthcdr(n, x) = cdr(nthcdr(n - 1, x)$$

$$nones(0, x) = x$$
$$nones(n, x) = 1 :: nones(n - 1, x)$$

Note that when n=0, this generalization simplifies to our original theorem. This theorem can be proved using induction on n by +1, where the base case is when $n \geq len(x)$. We also add the additional hypothesis $n \leq len(x)$ to the main theorem.

This induction scheme is a result of two factors. First, we always induct on n by +1. Second, we discover the base case during the same process that chooses a representative case. We will explain this process in more detail in the next section.

The induction step of this proof states that we can reduce the n^{th} step of the proof to the $n+1^{th}$ step. The base case is trivial because we know that $nthcdr(n, x)$ is an atom when $n \geq len(x)$[2].

Note that if we simplify this generalization for a concrete value of n, it will match a line in the proof for a list of length two shown above. For example, $nones(1, ones(nthcdr(1, x))) = ones1(nthcdr(1, x), nones(1, nil))$ simplifies to $1 :: ones(cdr(x)) = ones1(cdr(x), 1 :: nil)$. Similarly, if we instantiate this generalization with $n = 0$ or $n = 2$, it will simplify to the first and third lines of the proof shown above, respectively. In essence, this generalization captures what the $n+1$th line of the proof will look like.

This suggests the following technique for discovering generalizations: look at a proof for a specific case of the original theorem, and then try to find a function f such that $f(n)$ simplifies to the $n + 1^{th}$ line of the proof. If the case we choose is representative of all the cases, then our generalization will apply to the other cases as well. Note that the proof for the case where x is a list of length three would be identical to the case where x is length two, except that it would contain one additional line, described by the generalization above where $n = 3$.

Of course, this will only work for theorems where a proof of the n^{th} instance contains exactly $n + 1$ steps. Later on we will discuss how to deal with other types of theorems, but for now let's focus on how to find such an f when this criterion is met.

If we look at line three of the proof, we see that there are several places in the term where repetition occurs:

$$1 :: 1 :: ones(cdr(cdr(x))) = ones1(cdr(cdr(x)), 1 :: 1 :: nil)$$

[2]ACL2 can prove this theorem directly without any generalization.

The function *cons* is repeated twice at the top level of the left hand side. On the right hand side, it is also repeated twice inside the second argument to *ones*1. Similarly, the function *cdr* is repeated twice on the left hand side inside the function *ones* and it is repeated twice in the first argument to *ones*1 on the right hand side. Such repetitions may be indicative of a larger pattern throughout the proof. In order to indicate this, we can replace such repetitions by calls to functions that generate such repetitions. These calls will contain constants that indicate how many repetitions are present in the current formula, but at the same time, they also indicate how the current formula may be generalized because these constants can be replaced with expressions involving new constants or variables.

For example, consider replacing the nested *cdr*s with calls to the function *nthcdr*:

$$1 :: 1 :: ones(nthcdr(2, x)) = ones1(nthcdr(2, x), 1 :: 1 :: nil)$$

Although this is an equivalent formula, it is easy to see how it might be generalized by replacing the constant 2 with a variable n. Similarly, replacing the nested conses of ones yields:

$$nones(2, ones(nthcdr(2, x))) = ones1(nthcdr(2, x), nones(2, nil))$$

This formula is identical to the generalization we were seeking above, except that the constant 2 has replaced the variable n. Thus, we have found a technique for finding a generalization of the original formula: look at a proof of a special case of our theorem that we think is representative of the rest, then find a line of the proof that contains a formula with repetitions. Replace these repetitions with function calls to functions that repeat a given number of times. Finally, to generalize this formula, replace all integral constants with variables.

Clearly, although this technique works for the example presented above, it will need to be improved before it can be applied to many more complex theorems. In the rest of this section, we will discuss how to formalize the concepts presented here, as well as how to extend them for more complex scenarios.

2.2 Choosing a Representative Case

In the previous section, we were trying to prove $ones(x) = ones1(x, nil)$, and we started by considering the case when x was a list of length two. For an arbitrary theorem, how do we know what case to choose? For now, let us consider the case when only a single variable is present.

In the previous section, we presented two-part definitions for ones and ones1 for ease of presentation. Here, we present the definition of ones in a slightly different form to make our technique clear [3].

```
(defun ones (x)
  (if (endp x)
      nil
    (cons 1 (ones (cdr x)))))
```

This definition corresponds to the one in the previous section, but is presented in LISP format because it makes the branching condition explicit.

To find a suitable case, we look at the recursive calls present in the theorem. We choose an arbitrary call which contains that variable in an inductive position. In ACL2, each recursive function is admitted along with a measure for termination. The variables that are used in that measure determine which positions are inductive.

[3] endp tests for an empty list.

Once we have chosen a call, we unwind the call to a fixed depth and collect the conditions along the recursive branch until the final iteration, where we collect the condition leading to the base case. We conjoin these conditions to form a hypothesis that we use to generate a representative case for the theorem.

For example, in the theorem $ones(x) = ones1(x, nil)$, we could choose either call. We pick $ones(x)$ arbitrarily. Unwinding this function once gives:

```
(if (endp x)
      nil
    (cons 1 (ones (cdr x))))
```

and twice gives:

```
(if (endp x)
      nil
    (cons 1 (if (endp (cdr x))
                nil
              (cons 1 (ones (cddr x))))))
```

Collecting the condition from the recursive side of the first branch, along with the condition from the base side of the second branch and conjoining them gives `(and (not (endp x)) (endp (cdr x)))`. We can imagine unwinding three times would give us `(and (not (endp x)) (not (endp (cdr x))) (endp (cdr (cdr x))))`. This is the condition that represents a list of length two, which we can use to generate the proof of the finite case.

We can use our abstraction techniques to generalize such conditions. We will later explain in more detail how abstraction works, but for now we show the results of abstraction. If we abstract the condition of following the recursive branch

n times, we can generate the following function [4]:

```
(defun ncond (n x)
  (if (zp n)
      t
    (and (consp (nthcdr (- n 1) x))
         (ncond (- n 1) x))))
```

The call `(ncond n x)` represents the case corresponding to recurring n times. Note that `(ncond n x)` is true iff x has length at least n.

We can use this definition to form the induction scheme that will be used to prove our theorem:

```
(defun ifun (n x)
  (if (or (and (ncond n x)
               (not (consp (nthcdr n x))))
          (not (ncond n x)))
      nil
    (ifun (+ 1 n) x)))
```

The first disjunct in the base case, `(and (ncond n x) (not (consp (nthcdr n x))))`, represents recurring n times and then terminating. In this example, this corresponds to the condition $n = len(x)$. The second disjunct, `(not (ncond n x))`, corresponds to not having recurred n times, or the condition $n \geq len(x)$. Combined, we get $n \geq len(x)$. Thus, this scheme inducts on n by +1 with base case $n \geq len(x)$, which is exactly what we used in the previous section. We use the same procedure for generating induction schemes for all of the theorems in this

[4] Zp tests for zero.

section. We also use the condition `(ncond n x)` as a hypothesis to our theorem. Note that this condition is the same as $n \leq len(x)$. However, we do not actually prove this equality, we only use the more familiar form of this condition for clarity of explanation. Our actual implementation generates these condition in terms of ncond. When the condition $n \leq len(x)$ and the base case condition $n \geq len(x)$ are conjoined, we get the case $n = len(x)$. This is the condition will will be using for the base case of the examples later in this section. Since we technically are not using the function $len(x)$, in order to write $len(x)$, we use a variable k with the hypothesis `(ncond k x)` and the conditions from the base case of $ifun$, namely `(or (and (ncond k x) (not (consp (nthcdr k x)))) (not (ncond k x)))`. These two conditions ensure that k is equal to the length of x. For brevity, we will write $len(x)$ instead of k under these hypothesis. The examples we present will all be using this scheme. Rather than showing the complete hypotheses and base case each time, we instead present these examples without hypotheses and consider the base case to be when $n = len(x)$, using this familiar notation instead of the more accurate but harder to understand conditions based on ncond.

2.3 Formalization of Repetitions and Abstractions

We represent terms in the typical LISP fashion, using nested lists. A function call is represented by a list where the first element contains the symbol for the name of the function, and the remaining elements contain the arguments to the function. For example, the term `foo(cdr(cdr(x)), y)` is represented by the nested cons structure formed by evaluating the term:

```
cons('foo,
     cons(cons('cdr, cons(cons('cdr, cons('x, nil)), nil)),
```

```
cons('y, nil)))
```

Since this structure is made from conses, we can manipulate it using basic list processing primitives such as *car*, *cdr* and *cons*. Rewrite based proofs of cases are stored as lists of terms, using the same list processing primitives.

In order to describe the algorithm for generating generalizations, we will need a few definitions.

The function nth(n, x) returns the n^{th} element of x. We use x[p] to denote the subterm of x at path p. A path is a list of integers, used to index into a term. We can define x[p] as x when p is nil, otherwise x[p] is nth(car(p), x)[cdr(p)]. We say a path is valid for term x when computing x[p] does not call nth on an atom.

The function replace-at(p, y, x) returns the result of replacing the subterm located at path p in x with y. Let rep(p,n) be the path obtained by appending p with itself n times. Let REC be a new symbol not present in any other term, let m be an integer greater than 1, let p be a path in x, and let q be a path such that p @ rep(q,n)[5] is a path in x. If forall i : $0 \leq$ i $<$ m-1 : replace-at(q, REC, x[p@rep(q,i)]) = replace-at(q, REC, x[p@rep(q,i+1)]), then we call (p,q,m) a *repetition* in x. For example, if x is $cdr(cdr(cdr(x)))$, then (nil,[1],3) is a repetition in x.

Let (p,q,m) be a repetition in x. Let f be a new function symbol. Let f(0, a) = a and let f(n, a) be defined by replacing the subterm at path q in x[p] with the recursive call f(n-1, a). For example, if (p,q,m) is (nil,[1],3) and x is cdr(cdr(cdr(x))) then f(n,a) would be defined as cdr(f(n-1,a)). We call f(m,x[p@rep(q,m)]) a *simple abstraction* of x[p]. A simple abstraction of a term is an equivalent term, but it hints at a generalization for that term. For example, nthcdr(3,x) is a simple abstraction of cdr(cdr(cdr(x))).

[5] @ is infix append.

In order to generalize a term, we must replace all repetitions with simple abstractions. We can do this using a preorder traversal of the subterms of the term, replacing each repetition as we encounter it. One subtlety to this approach is that there may be nested repetitions. For example, the first repetition during a preorder traversal of $1 :: 1 :: ones(cdr(cdr(x)))$ is (nil, [2], 2). Replacing this repetition with its abstraction yields $nones(2, ones(cdr(cdr(x))))$. In order to complete the abstraction, we must next process the term $ones(cdr(cdr(x)))$. When this process is complete, we have formed a *full abstraction* of x.

2.4 Generating Proofs for Cases

In the previous example, we made the assumption that the proof of the case we choose was of the form:

$$ones(x) = ones1(x, nil)$$
$$1 :: ones(cdr(x)) = ones1(cdr(x), 1 :: nil)$$
$$1 :: 1 :: ones(cdr(cdr(x))) = ones1(cdr(cdr(x)), 1 :: 1 :: nil)$$
$$1 :: 1 :: nil = 1 :: 1 :: nil$$

In fact, creating a rewriter to generate such a proof would be difficult because knowing when to open a definition on the lhs and when to open one on the rhs in order to synchronize the two sides would be difficult. A typical inside-out rewriter would instead generate a proof such as this:

$$ones(x) = ones1(x, nil)$$

$$1 :: ones(cdr(x)) = ones1(x, nil)$$

$$1 :: 1 :: ones(cdr(cdr(x))) = ones1(x, nil)$$

$$1 :: 1 :: nil = ones1(x, nil)$$

$$1 :: 1 :: nil = ones1(x, nil)$$

$$1 :: 1 :: nil = ones1(cdr(x), 1 :: nil)$$

$$1 :: 1 :: nil = ones1(cdr(cdr(x)), 1 :: 1 :: nil)$$

$$1 :: 1 :: nil = 1 :: 1 :: nil$$

Note that there is no single line of this proof that is representative of the entire proof, as there was in the proof above. This is because the proof is performed in two stages, first for the lhs and then for the rhs. Consequently, we can no longer apply the technique we described above for generalizing this theorem.

In order to solve this problem, we split the proof into two parts, one for the rhs and one for the lhs. We write the proof using a new *factored* notation:

$$\begin{bmatrix} \text{ones(x)} \\ \text{1::ones(cdr(x))} \\ \text{1::1::ones(cdr(cdr(x)))} \\ \text{1::1::nil} \end{bmatrix} = \begin{bmatrix} \text{ones1(x, nil)} \\ \text{ones1(cdr(x), 1::nil)} \\ \text{ones1(cdr(cdr(x)), 1::1::nil)} \\ \text{1::1::nil} \end{bmatrix}$$

We store this proof using a nested list structure just as before. The only difference is that instead of the proof being a list of terms as before, terms can contain proofs. In this case, the proof is of the form $lhs = rhs$ where lhs and rhs are both lists of terms.

The lhs of this structure is a proof for the theorem $ones(x) = 1 :: 1 :: nil$ and the rhs is a proof for the theorem $ones1(x, nil) = 1 :: 1 :: nil$. Since the rhss of each

of these equalities are the same, we can combine them to form the original equality $ones(x) = ones1(x, nil)$. Although this proof applies only for the case when x is a list of length two, we can prove the general case using a similar technique after generalizing each of these two lemmas.

First we apply the generalization technique we described earlier to each of the two sides. During this step, we cache definitions, avoiding distinct isomorphic definitions. This gives:

$$nones(n, ones(nthcdr(n, x)))$$

for the lhs and

$$ones1(nthcdr(n, x), nones(n, x))$$

for the rhs. Note that simplifying either of these formulas for a particular n produces the $n+1^{th}$ line of the respective side of the proof. Thus, if we want to form a lemma that equates the first and last step of the lhs, we can replace n with 0 and equate it to another copy of the generalization where n is replaced with $len(x)$. We know that $len(x)$ is the final step, because that is the condition we derived for the base case of the induction, as described in Section 2.2. On the lhs, this gives:

$$nones(0, ones(nthcdr(0, x))) = nones(len(x), ones(nthcdr(len(x), x))) \qquad (2.1)$$

which simplifies to [6]:

$$ones(x) = nones(len(x), nil)$$

[6] assuming the lemma $nthcdr(len(x), x)) = nil$, which we discover by attempting to equate the RHSs of Eqs 2.1 and 2.2, can be proven by ACL2 without any generalization

Similarly, on the rhs, we get:

$$ones1(nthcdr(0,x),nones(0,nil)) = ones1(nthcdr(len(x),x),nones(len(x),nil)) \tag{2.2}$$

which simplifies [7] to:

$$ones1(x,nil) = nones(len(x),nil)$$

Since the rhs of each of these lemmas is the same, we can combine them to prove the original theorem as before.

We can prove Lemmas 2.1 and 2.2 by replacing 0 with n and inducting on $n+1$ with base case $n \geq len(x)$. This is the same induction scheme we used in Section 2.1.

In the next section, we will present another technique for handling more complex examples. Later, we will formalize how we generate ACL2 proofs from these abstractions.

2.5 Integer Deltas

The key step for finding generalizations using the technique presented has been to replace repetitions with abstractions. So far we have only considered the case where each repetition is identical to the previous. However, there are times when each repetition varies slightly from the previous in a predictable way. Consider the term $cons(3,cons(2,cons(1,nil)))$. We can abstract this term using the following function:

$$f(0) = nil$$

[7] again assuming $nthcdr(len(x),x) = nil$

$$f(n) = cons(n, f(n-1))$$

Using this function, we can represent the above term as $f(3)$. Notice, however, that the repetition above does not meet the criteria for a repetition that we introduced earlier because each repetition is slightly different from the previous. If we were trying instead to abstract $cons('foo, cons('bar, cons('baz, nil)))$ it would have been more difficult.

Let's modify our original definition for repetitions and abstractions to accommodate this new type of pattern.

If x and y are two terms and x[p] and y[p] have different top function symbols, then we call (p,x[p],y[p]) a *difference* between x and y. If for a given difference (p, x[p], y[p]) and for all paths q which are strictly prefixes of p, there are no differences between the top function symbols at path q in x and at path q in y, then (p,x[p],y[p]) is a *maximal difference*.

Let diff(x,y) be the list of all maximal differences between x and y. We say that a difference (p, x[p], y[p]) is an *integer difference* iff x[p] and y[p] are integers.

If all differences in diff(x,y) are integer differences, let deltas(x,y) be the list of pairs (p,d), called *deltas*, such that d=x[p]-y[p] for each difference (p, x[p], y[p]) in diff(x,y).

Now, let's redefine repetitions and abstractions using these new ideas. Let REC be a new symbol not present in any other term, let n be an integer greater than 2, let p be a path in x, and let q be a path such that p @ rep(q,n) is a path in x. If the calls to *deltas* below are well defined (i.e., the underlying differences are integer differences), and the following condition is satisfied, then we call (p,q,n) a

repetition in x:

$$\begin{aligned}
&\forall i : 0 \leq i < n\text{-}2 : \\
&deltas(replace\text{-}at(q, REC, x[p@rep(q, i)]), \\
&\qquad replace\text{-}at(q, REC, x[p@rep(q, i+1)])) \\
&= \\
&deltas(replace\text{-}at(q, REC, x[p@rep(q, i+1)]), \\
&\qquad replace\text{-}at(q, REC, x[p@rep(q, i+2)]))
\end{aligned}$$

For example, if x is $cons(3, cons(2, cons(1, nil)))$ and p is nil, we have that rep([2], 3) is a path in x. For $i = 0$, there are three main things we must compute:

$$\begin{aligned}
&replace\text{-}at([2], REC, x[rep([2], 0)]) \\
&replace\text{-}at([2], REC, x[rep([2], 1)]) \\
&replace\text{-}at([2], REC, x[rep([2], 2)])
\end{aligned}$$

for the given x, these three values are:

$$\begin{aligned}
&cons(3, REC) \\
&cons(2, REC) \\
&cons(1, REC)
\end{aligned}$$

We refer to these terms as *step terms*. The definition above requires the deltas between the first two step terms to be equal to the deltas between the last two. In this case, both sets of deltas are [([1],1)]. Thus, (nil,[2],3) is a repetition in x.

Let (p,q,m) be a repetition in x. Let f be a new function symbol. Let f(0, a) = a and let f(n, a) be defined by replacing the subterm at path q in x[p] with the recursive call f(n-1, a). For example if (p,q,m) is (nil,[2],3) and x

is $cons(3, cons(2, cons(1, nil)))$ then f(n,a) would be defined as $cons(3, f(n\text{-}1, a))$. We will further modify this definition as follows. Let D be deltas(replace-at(q, REC, x[p@rep(q,0)]), replace-at(q, REC, x[p@rep(q,1)])). For each delta (r, d) in D, replace the term at path r in the definition of f(n,a) with the term $d * (n\text{-}1) + x[p@rep(q, m\text{-}1)@r]$, where n is a symbol and the rest of the expression is simplified using the given values. As before, we call f(m,x[p@rep(q,m)]) a *simple abstraction* of x[p]. Let's complete the example above where (nil,[2],3) is a repetition in x, and x is $cons(3, cons(2, cons(1, nil)))$. We first define f(0, a) = a and f(n, a) = cons(3, f(n-1,a)). Notice how the definition contains the constant 3. We need to modify this part of the definition to decrease as n decreases. We know from above that the delta between the first two step terms is ([1], 1). Thus, we replace the term at path [1] in cons(3, f(n-1,a)) with $1 * (n\text{-}1) + x[nil@rep([2], 2)@[1]]$. The term $x[nil@rep([2], 2)@[1]]$ refers to the value of the subterm that changes in the final step term. In this example, this subterm starts at 3 and decreases to 1. Thus, this value is 1. Simplifying the remainder of the term simply gives n. Replacing the constant 3 in the definition above with n gives us the definition f(n, a) = cons(n, f(n-1, a)), which corresponds exactly to the example we used to generate this definition. Once we have generated this definition, we can use it to abstract the original example. Here, we call f(3, nil) a *simple abstraction* of $cons(3, cons(2, cons(1, nil)))$. We also refer to f as a *pattern function*.

2.5.1 Extracting Constants

Consider for a moment if we were to use the above technique to abstract the term $cons(0, cons(1, cons(2, nil)))$. Following the steps above, we would generate a function g such that $g(0, a) = a$ and $g(n, a) = cons(3\text{-}n, g(n\text{-}1, a))$. Instead of storing

g as shown, we extract the constants and add additional variables to the function before storing it in our table of abstractions. In this case, g would become $g(n, m, a) = cons(m\text{-}n, g(n\text{-}1, a))$. We don't extract the 1 from n-1 because that is how all of our abstractions recur. The purpose of this extraction is to prevent a proliferation of pattern functions. Since we check to see if a new pattern function is isomorphic to any other pattern function before introducing a new function symbol, we can avoid generating functions that vary only by a constant.

2.6 Term Deltas

When the changes between terms in a repetition are integral, it is easy to compute a delta. However, there are many repetitions that are not captured by integer deltas. Consider the following term[8]:

$$cons(car(x), cons(cadr(x), cons(caddr(x), nil)))$$

We can see that there is a consistent change between the elements of the list. If we were somehow to rewrite the above term as

$$cons(car(nthcdr(0, x)), cons(car(nthcdr(1, x)), cons(car(nthcdr(2, x)), nil)))$$

we could use integer deltas to abstract this term. Although this term can be formed by abstracting the subterms corresponding to the calls of nthcdr, it is difficult to compute abstractions for repetitions that occur zero or one times, as is the case with $car(x)$ and $cadr(x)$. This is because there are no constraints on an abstraction that is valid for zero repetitions; almost any function can be repeated zero times to form an equivalent term. A similar problem occurs with a single repetition.

[8] cadr(x) is an abbreviation for car(cadr(x)), similarly for caddr(x) and car(cdr(cdr(x)))

We instead solve this problem in another way. We compute *term deltas* for the sequence. If these deltas are consistent, then we will use them to generate abstractions for the subterms and then use integer deltas to abstract the full term.

Here is how this technique works. We first look for a *potential repetition*, which is a repetition where we have only verified that the top function symbol is the same for each term in the sequence. When we find the potential repetition (nil,[2],3) corresponding to the three nested conses, we first compute the sequence of *step terms*:

$$cons(car(x), REC)$$
$$cons(cadr(x), REC)$$
$$cons(caddr(x), REC)$$

Next, we compare each step term to the next as follows. First, we remove the parts of the terms that match exactly. This leaves:

$$x$$
$$cdr(x)$$
$$cddr(x)$$

Now, we look for each term in the next term. Thus 'x' is found inside cdr(x), and cdr(x) is found inside cdr(cdr(x)). Since we found a match, we generate a pattern function call that repeats the shell of the outside term at the position where the subterm was found. In this case, this results in a function that repeats cdrs inside each other. This function is commonly known as nthcdr. Replacing each term with a call to the generated function *nthcdr* yields:

$$nthcdr(0, x)$$
$$nthcdr(1, x)$$

$$nthcdr(2, x)$$

Now that these subterms have been abstracted, they are substituted back into the original term to form the term we were hoping for:

$$cons(car(nthcdr(0, x)), cons(car(nthcdr(1, x)), cons(car(nthcdr(2, x)), nil)))$$

This term is abstracted using integer deltas and constant extraction. The result is $ncons(3, 3, x, nil)$, where $ncons$ is defined as:

$$ncons(0, m, x, a) = a$$
$$ncons(n, m, x, a) = cons(car(nthcdr(m\text{-}n, x)), ncons(n\text{-}1, m, x, a))$$

With this more powerful abstraction mechanism we will be able to tackle more complex theorems.

2.7 Nested Patterns

Consider the term $ap(ap(x, x), x) = ap(x, ap(x, x))$. Note that this theorem cannot be proved directly by induction. Typically it is generalized to $ap(ap(y, x), x) = ap(y, ap(x, x))$. We will show how our technique can automatically find an alternative generalization.

Let us first focus on the term $ap(x, x)$. If we rewrite this term under the assumption x is length 3, we get:

$$ap(x, x)$$
$$cons(car(x), ap(cdr(x), x))$$
$$cons(car(x), cons(cadr(x), ap(cddr(x), x)))$$
$$cons(car(x), cons(cadr(x), cons(caddr(x), ap(cdddr(x), x))))$$
$$cons(car(x), cons(cadr(x), cons(caddr(x), x)))$$

If we look at the 4th line in this sequence, we see a pattern very similar to the one in the preceding section. One difference is that there is a $cdddr(x)$ on the inside instead of a nil. This is what we consider a nested pattern. To deal with such patterns, we first abstract the outer pattern, yielding:

$$ncons(3, 3, x, ap(cdddr(x), x))$$

where $ncons$ is defined as:

$$ncons(0, m, x, a) = a$$
$$ncons(n, m, x, a) = cons(car(nthcdr(m\text{-}n)), ncons(n\text{-}1, m, x, a))$$

then we abstract the inner pattern:

$$ncons(3, 3, x, ap(nthcdr(3, x), x))$$

If we try to fit this pattern to lines two and three, we would see they follow a similar patter, where n=1 and n=2. This tells us how the numbers vary with respect to the step of the proof. This lets us form the final abstraction:

$$ncons(n, n, x, ap(nthcdr(n, x), x))$$

We will repeat this process for a variety of proof sequences below.

Now, we will look at the entire proof for $ap(ap(x, x), x) = ap(x, ap(x, x))$ and use this function to abstract the proof.

If we generate a proof for the case where x is a list of length two, we get the

following:

$$ap(ap(x,x),x) =$$
$$ap(x,ap(x,x))$$

$$ap(cons(car(x),ap(cdr(x),x)),x) =$$
$$ap(x,ap(x,x))$$

$$ap(cons(car(x),cons(cadr(x),ap(cddr(x),x))),x) =$$
$$ap(x,ap(x,x))$$

$$ap(cons(car(x),cons(cadr(x),x)),x) =$$
$$ap(x,ap(x,x))$$

$$cons(car(x),ap(cons(cadr(x),x),x)) =$$
$$ap(x,ap(x,x))$$

$$cons(car(x),cons(cadr(x),ap(x,x))) =$$
$$ap(x,ap(x,x))$$

$$cons(car(x),cons(cadr(x),cons(car(x),ap(cdr(x),x)))) =$$
$$ap(x,ap(x,x))$$

$$cons(car(x),cons(cadr(x),cons(car(x),cons(cadr(x),ap(cddr(x),x))))) =$$
$$ap(x,ap(x,x))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$ap(x, ap(x, x))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$cons(car(x), ap(cdr(x), ap(x, x)))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$cons(car(x), cons(cadr(x), ap(cddr(x), ap(x, x)))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$cons(car(x), cons(cadr(x), ap(x, x)))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$cons(car(x), cons(cadr(x), cons(car(x), ap(cdr(x), x))))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), ap(cddr(x), x)))))$$

$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x)))) =$$
$$cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x))))$$

If we split this proof along the lhs and rhs, as we did in the previous section, we get

the following:

$$\left[\begin{array}{l}
1\ ap(ap(x,x),x) \\
2\ ap(cons(car(x),ap(cdr(x),x)),x) \\
3\ ap(cons(car(x),cons(cadr(x),ap(cddr(x),x))),x) \\
4\ ap(cons(car(x),cons(cadr(x),x)),x) \\
5\ cons(car(x),ap(cons(cadr(x),x),x)) \\
6\ cons(car(x),cons(cadr(x),ap(x,x))) \\
7\ cons(car(x),cons(cadr(x),cons(car(x),ap(cdr(x),x)))) \\
8\ cons(car(x),cons(cadr(x),cons(car(x),cons(cadr(x),ap(cddr(x),x))))) \\
9\ cons(car(x),cons(cadr(x),cons(car(x),cons(cadr(x),x))))
\end{array}\right]$$

$$=$$

$$\left[\begin{array}{l}
1\ ap(x,ap(x,x)) \\
2\ cons(car(x),ap(cdr(x),ap(x,x))) \\
3\ cons(car(x),cons(cadr(x),ap(cddr(x),ap(x,x)))) \\
4\ cons(car(x),cons(cadr(x),ap(x,x))) \\
5\ cons(car(x),cons(cadr(x),cons(car(x),ap(cdr(x),x)))) \\
6\ cons(car(x),cons(cadr(x),cons(car(x),cons(cadr(x),ap(cddr(x),x))))) \\
7\ cons(car(x),cons(cadr(x),cons(car(x),cons(cadr(x),x))))
\end{array}\right]$$

Although splitting the proof has improved its readability somewhat, this proof does not have some of the important properties that the example in Section 2.4 had. In the proof about *ones*, the lhs had exactly 3 steps when we considered a list of length 2: two to open the recursive case of *ones*, then one for the base case. When we abstracted that proof, we discovered a generalization that represented opening the recursive function n times and then applying the base case. One reason

this worked is that a single line of the proof captured the entire proof. In this example, there are multiple recursive functions being opened up. This makes it difficult to see how a single formula could represent the entire lhs as a function of the size of the input list.

Instead of trying to find a single formula to fit the entire proof, we will break it up into sections and fit a formula to each section. Each section represents the expansion of a particular recurring function, and results in a lemma proved by induction in the same way as the examples in Sections 2.1 and 2.4. Later on we will introduce methods that allow us to find these sections automatically.

The first section on the lhs of the proof can be abstracted using ncons in the same way as above:

$$
\begin{aligned}
&1\ ap(ap(x,x),x)\\
&2\ ap(cons(car(x),ap(cdr(x),x)),x)\\
&3\ ap(cons(car(x),cons(cadr(x),ap(cddr(x),x))),x)\\
&4\ ap(cons(car(x),cons(cadr(x),x)),x)
\end{aligned}
$$

becomes:

$$
\begin{aligned}
&ap(ncons(0,0,x,ap(nthcdr(0,x),x)),x)\\
&ap(ncons(1,1,x,ap(nthcdr(1,x),x)),x)\\
&ap(ncons(2,2,x,ap(nthcdr(2,x),x)),x)\\
&ap(ncons(2,2,x,x),x)
\end{aligned}
$$

After generalizing the integers in this section, we get the following lemma:

$$
\begin{aligned}
ap(ncons(n,n,x,ap(nthcdr(n,x),x)),x) = \\
ap(ncons(len(x),len(x),x),x)
\end{aligned}
\tag{2.3}
$$

The LHS of this lemma is obtained by abstracting the first three lines of the proof. The RHS of this lemma is obtained by abstracting the final line and substituting $len(x)$ for n. We know to substitute $n = len(x)$ because that is the base case of the induction scheme for this theorem. For a detailed discussion of how we generated this scheme, see section 2.2. In essence, this lemma is used to prove that an arbitrary line of the proof is equal to the final line. This lemma can be proved by induction, using the same scheme we used in Sections 2.1 and 2.4.

The next section also can be represented using *ncons*, except that the base case differs from the previous terms. Notice that the inner calls of *ncons* all have the same value for the second parameter of *ncons*. This is what allows *ncons* to represent lists that start with elements other than the first in the list.

$$4\ ap(cons(car(x), cons(cadr(x), x)), x)$$
$$5\ cons(car(x), ap(cons(cadr(x), x), x))$$
$$6\ cons(car(x), cons(cadr(x), ap(x, x)))$$

becomes:

$$ncons(0, 0, x, ap(ncons(2, 2, x, x), x))$$
$$ncons(1, 1, x, ap(ncons(1, 2, x, x), x))$$
$$ncons(2, 2, x, ap(ncons(0, 2, x, x), x))$$

This section yields the lemma:

$$ncons(n, n, x, ap(ncons(len(x)\text{-}n, len(x), x, x), x)) = \tag{2.4}$$
$$ncons(len(x), len(x), x, ap(ncons(0, len(x), x, x), x)) \tag{2.5}$$

In this case, the integer deltas from Section 2.5 yield the term $len(x)$-n for the ascending sequence in the inner ncons. This is just a way to make mathematically

precise what we can easily see: that each number is one less than the previous in the sequence.

The final sequence on the lhs removes the final call of *ap*:

$$6\ cons(car(x), cons(cadr(x), ap(x, x)))$$
$$7\ cons(car(x), cons(cadr(x), cons(car(x), ap(cdr(x), x))))$$
$$8\ cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), ap(cddr(x), x)))))$$
$$9\ cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x))))$$

which abstracts to:

$$ncons(2, 2, x, ncons(0, 0, x, ap(nthcdr(0, x), x)))$$
$$ncons(2, 2, x, ncons(1, 1, x, ap(nthcdr(1, x), x)))$$
$$ncons(2, 2, x, ncons(2, 2, x, ap(nthcdr(2, x), x)))$$
$$ncons(2, 2, x, ncons(2, 2, x, x))$$

which gives:

$$ncons(len(x), len(x), x, ncons(n, n, x, ap(nthcdr(n, x), x))) = \tag{2.6}$$
$$ncons(len(x), len(x), x, ncons(len(x), len(x), x, x))$$

We always instantiate our lemmas with n=0 in order to chain them together. This is because the lemmas are designed to equate the n^{th} line of the proof with the final line. If we instantiate Eqns. 2.3, 2.4 and 2.6 with $n = 0$ and simplify, we can combine them by equating the rhs of the first and the lhs of the second, along with the rhs of the second and the lhs of the third. This gives a combined lemma for the

lhs:

$$ap(ap(x,x),x) = \tag{2.7}$$
$$ncons(len(x),len(x),x,ncons(len(x),len(x),x,x))$$

On the rhs we have two distinct sequences.

1 $ap(x,ap(x,x))$

2 $cons(car(x),ap(cdr(x),ap(x,x)))$

3 $cons(car(x),cons(cadr(x),ap(cddr(x),ap(x,x))))$

4 $cons(car(x),cons(cadr(x),ap(x,x)))$

becomes:

$$ncons(0,0,x,ap(nthcdr(0,x),ap(x,x)))$$
$$ncons(1,1,x,ap(nthcdr(1,x),ap(x,x)))$$
$$ncons(2,2,x,ap(nthcdr(2,x),ap(x,x)))$$
$$ncons(2,2,x,ap(x,x))$$

This generalizes to:

$$ncons(n,n,x,ap(nthcdr(n,x),ap(x,x))) = \tag{2.8}$$
$$ncons(len(x),len(x),x,ap(x,x))$$

The final sequence:

4 $cons(car(x), cons(cadr(x), ap(x, x)))$

5 $cons(car(x), cons(cadr(x), cons(car(x), ap(cdr(x), x))))$

6 $cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), ap(cddr(x), x)))))$

7 $cons(car(x), cons(cadr(x), cons(car(x), cons(cadr(x), x))))$

becomes:

$$ncons(2, 2, x, ncons(0, 0, x, ap(nthcdr(0, x), x)))$$
$$ncons(2, 2, x, ncons(1, 1, x, ap(nthcdr(1, x), x)))$$
$$ncons(2, 2, x, ncons(2, 2, x, ap(nthcdr(2, x), x)))$$
$$ncons(2, 2, x, ncons(2, 2, x, x))$$

Which gives:

$$ncons(len(x), len(x), x, ncons(n, n, x, ap(nthcdr(n, x), x))) = \quad (2.9)$$
$$ncons(len(x), len(x), x, ncons(len(x), len(x), x, x))$$

If we instantiate both Eqs. 2.8 and 2.9 with $n = 0$ and simplify, we can equate the rhs of Eq. 2.8 and the lhs of Eq. 2.9. Chaining together these two equalities, we form a single equality for the entire rhs:

$$ap(x, ap(x, x)) = \quad (2.10)$$
$$ncons(len(x), len(x), x, ncons(len(x), len(x), x, x))$$

Notice that the rhss of Eqs. 2.7 and 2.10 are the same, the the lhss are the original formulas we were trying to equate. Thus, we have completed the proof.

2.8 Multiple Variables

We have begun investigation into extending our technique to deal with theorems that have multiple variables. At present, we can handle the case where the interaction between multiple variables can be described by addition. We believe these techniques can be extended to more general types of interactions, but present only addition here.

Consider the theorem

$$\text{nthcdr(x,nthcdr(y,z))} = \text{nthcdr(y,nthcdr(x,z))},$$

where *nthcdr* is defined as:

$$nthcdr(0, y) = y$$
$$nthcdr(n, y) = nthcdr(n\text{-}1, cdr(y)))$$

If we rewrite the lhs for the case where x is 3, we get:

$$nthcdr(x, nthcdr(y, z))$$
$$nthcdr(x\text{-}1, cdr(nthcdr(y, z)))$$
$$nthcdr(x\text{-}2, cdr(cdr(nthcdr(y, z))))$$
$$nthcdr(x\text{-}3, cdr(cdr(cdr(nthcdr(y, z)))))$$
$$nthcdr(0, cdr(cdr(cdr(nthcdr(y, z)))))$$
$$cdr(cdr(cdr(nthcdr(y, z))))$$

Using the function f, defined below, we can abstract the proof above as follows:

$$f(0, a) = a$$
$$f(n, a) = cdr(f(n\text{-}1, a))$$

$$nthcdr(x, nthcdr(y, z))$$
$$nthcdr(x\text{-}1, f(1, nthcdr(y, z)))$$
$$nthcdr(x\text{-}2, f(2, nthcdr(y, z)))$$
$$nthcdr(x\text{-}3, f(3, nthcdr(y, z)))$$
$$nthcdr(0, f(3, nthcdr(y, z)))$$
$$f(3, nthcdr(y, z))$$

This suggests the general formula $f(n, nthcdr(y, z))$ for the final result. If we apply the same technique to the rhs, we get $nthcdr(y, f(n, z))$. Since the two sides are not equal, our proof will fail. Notice the main reason that we have not attained a normal form is that we did not choose a case that allows us to expand the calls to $nthcdr$ on y. If we instead choose a case where both x and y are 3, then we can expand those calls too:

$$nthcdr(x, nthcdr(y, z))$$
$$nthcdr(x, nthcdr(y\text{-}1, cdr(z)))$$
$$nthcdr(x, nthcdr(y\text{-}2, cdr(cdr(z))))$$
$$nthcdr(x, nthcdr(y\text{-}3, cdr(cdr(cdr(z)))))$$
$$nthcdr(x, cdr(cdr(cdr(z)))))$$
$$nthcdr(x\text{-}1, cdr(cdr(cdr(cdr(z))))))$$
$$nthcdr(x\text{-}2, cdr(cdr(cdr(cdr(cdr(z)))))))$$
$$nthcdr(x\text{-}3, cdr(cdr(cdr(cdr(cdr(cdr(z))))))))$$
$$cdr(cdr(cdr(cdr(cdr(cdr(z)))))))$$

Now we have a new problem. The final step abstracts to $f(6, z)$. Normally, when we generalize a formula we simply replace the constants with a single variable. However, since the case we chose depended on both x and y, we will not get a good

generalization by using a single variable. Ideally, we want to replace the constant 6 with an expression of x and y, in this case, $x + y$. However, the formula $f(6, z)$ does not suggest any way to discover this expression.

We solve this problem by computing the abstraction in three different ways. First, we compute a baseline abstraction where x and y are both 3. In this example, that gives us $f(6, z)$. Then, for each variable in our case, we compute an abstraction for a case where that variable has changed by one step and all other variables remain the same. In this example, we first compute an abstraction for when x is 4 and y remains 3, then we compute an abstraction for when x remains 3 and y is 4. In both of these cases, we get $f(7, z)$. What this reveals is that both variables are responsible for incrementing the number of iterations of f. From this, we can deduce that a good generalization of this abstraction is $f(n + m)$, where n represents the current case for x and m represents the current case for y.

2.9 Hybrid Proof Terms

In this section, we introduce a new datatype that we use to make recognizing more complex patterns easier. Although the example we present in this section could have been done without this new datatype, for purposes of exposition we have chosen a simple example. For a more complex theorem, such as $rev(x) = rev1(x, nil)$, this new datatype would be necessary. This is because rev is a recursive function that contains calls to another recursive function ap. These nested recursions form more complex patterns that necessitate the following data structure.

Consider the theorem $len(x) = len1(x, 0)$, where the definitions for len and $len1$ are given below. Note that in this section we use the notation $h :: t$ for $cons(h, t)$.

$$len(nil) = 0$$
$$len(h :: t) = 1 + len(t)$$

$$len1(nil, a) = a$$
$$len1(h :: t, a) = len1(t, 1 + a)$$

In the theorem above, the second argument in the call to $len1$ is the constant 0, while the definition for $len1$ modifies its second argument when it recurs. This problem is typically referred to as a blocked accumulator, and makes the theorem impossible to prove directly by induction. In order to solve this problem, the theorem must be generalized. The typical generalization is $a + len(x) = len1(x, a)$. This generalization can be proved by induction, but such generalizations may be difficult to discover automatically.

Our technique avoids this problem by finding a proof based on the patterns found in a proof of a finite instance of this theorem. Consider a proof of this theorem where x is a list of two elements.

$$len(x1 :: x2 :: nil) = len1(x1 :: x2 :: nil, 0)$$
$$1 + len(x2 :: nil) = len1(x2 :: nil, 1 + 0)$$
$$1 + 1 + len(nil) = len1(nil, 1 + 1 + 0)$$
$$1 + 1 + 0 = 1 + 1 + 0$$

Before we attempt to find patterns in this proof, we convert it to a form that is a hybrid of terms and proofs. It factors the terms according to the subterms that are modified during each step. The proof above can be rewritten using this representation as follows.

$$\left[\begin{array}{c} len(x_1 :: x_2 :: nil) \\ 1 + \left[\begin{array}{c} len(x_2 :: nil) \\ 1 + \left[\begin{array}{c} len(nil) \\ 0 \end{array}\right] \end{array}\right] \end{array}\right] = \left[\begin{array}{c} len1(x_1 :: x_2 :: nil, 0) \\ len1(x_2 :: nil, 1 + 0) \\ len1(nil, 1 + 1 + 0) \\ 1 + 1 + 0 \end{array}\right]$$

Notice how parts of the term that do not change from one line to the other are factored out and not repeated. For example, the top level equality is written only once. This also makes the patterns in the proof more apparent.

Next, we replace any simple patterns we find with recursive calls that mimic those patterns. The first of these patterns is a list containing elements of x. The second of these patterns is a sequence of nested additions by one.

$$\left[\begin{array}{c} len(f1(2, x)) \\ 1 + \left[\begin{array}{c} len(f1(1, x)) \\ 1 + \left[\begin{array}{c} len(f1(0, x)) \\ 0 \end{array}\right] \end{array}\right] \end{array}\right] =$$

$$\left[\begin{array}{c} len1(f1(2, x), f2(0)) \\ len1(f1(1, x), f2(1)) \\ len1(f1(0, x), f2(2)) \\ f2(2) \end{array}\right]$$

The definitions for $f1$ and $f2$ are below. The function $f1$ represents the nth tail of x. The function $f2$ represents n ones added to a zero. Note that these are simple functions that modify only one argument, n, recur on n-1, and terminate when n is zero. All pattern functions we introduce follow the same schema, although

they may have any number of additional non-inductive arguments and their bodies may vary [9].

$$f1(n, x) = f1'(n, n, x)$$
$$f1'(0, k, x) = nil$$
$$f1'(n, k, x) = car(nthcdr(k\text{-}n + 1, x)) :: f1'(n\text{-}1, k, x)$$

$$f2(0) = 0$$
$$f2(n) = 1 + f2(n\text{-}1)$$

Now that we have simplified the proof with these pattern functions, new simple patterns emerge. On the left hand side, we have nested calls to $len(f1(?, x))$ that expand and then recur. On the right hand side, we have a sequence of calls to $len1(f1(?, x), f2(?))$. The only things that change along these sequences are the particular integers used; the terms are structurally equivalent throughout the sequence. Thus, we can introduce the following two new pattern functions. Note that here we use the notation [a; b; c] to denote a list with three elements. If we write a list vertically, as we did for the proofs on the previous page, we will omit the semicolons. Please note that these are not functions in the ACL2 logic and they do not have a semantics. Instead, think of them as formal terms that represent proofs. Later, we will extract the top and bottom terms from these functions to form actual functions in the logic.

[9] In the definition for $f1$ below, we show nil as a base case. This is fine as long as we assume that x is a true-listp. In our actual theorems, we use a base case that corresponds to the actual atom at the end of the list. However, for ease of presentation we will treat all lists as true-lists and thus use a nil in this definition.

$$f3(0,x) = [len(f1(0,x)); 0]$$
$$f3(n,x) = [len(f1(n,x)); 1 + f3(n\text{-}1,x)]$$

$$f4(n,x) = f4'(n,n,x)$$
$$f4'(0,k,x) = [len1(f1(n,x), f2(k\text{-}n)); f2(k)]$$
$$f4'(n,k,x) = len1(f1(n,x), f2(k\text{-}n)) :: f4'(n\text{-}1,k,x)$$

Using these two new functions, the proof is reduced one last time.

$$f3(2,x) = f4(2,x)$$

To see how these functions represent the original proof, let us expand them one step at a time. We will start with the left hand side. This is just reversing the process above for illustrative purposes:

$$f3(2,x)$$

expands to

$$[len(f1(2,x)); 1 + f3(1,x)]$$

which, when written vertically, becomes:

$$\begin{bmatrix} len(f1(2,x)) \\ 1 + f3(1,x) \end{bmatrix}$$

expanding $f3$ again gives:

$$\begin{bmatrix} len(f1(2,x)) \\ 1 + \begin{bmatrix} len(f1(1,x)) \\ 1 + f3(0,x) \end{bmatrix} \end{bmatrix}$$

finally, expanding the final step of $f3$:

$$\left[\begin{array}{c} len(f1(2,x)) \\ 1+\left[\begin{array}{c} len(f1(1,x)) \\ 1+\left[\begin{array}{c} len(f1(0,x)) \\ 0 \end{array}\right] \end{array}\right] \end{array}\right]$$

By examining another instance of this theorem, say for a list of three elements, we can discover that the general form of the proof is $f3(n,x) = f4(n,x)$, where n is the length of x.

In order to generate a proof of our original theorem from this generalized proof instance, we must extract a sequence of lemmas. Each pattern function that contains proof steps is converted into a proof that its *top term* is equivalent to its *bottom term*. This process may involve first generating proofs for functions nested inside this function. The *top term* of a pattern function is obtained by taking the first term of every list in its body; the *bottom term* is obtained similarly but it uses the last term instead. If the first or last term is a pattern function, a new recursive function must be introduced to represent the top or bottom term, respectively. The base case for the top or bottom term is derived from the base case of its pattern function.

For the function $f3(n,x)$, we obtain the top term $len(f1(n,x))$ and the bottom term $f2(n)$. Typically we would generate a new recursive function for the bottom term, but in this case it would be identical to $f2$, so we reuse that definition.

To see how we generate these terms, recall the definition of $f3$:

$$f3(0,x) = [len(f1(0,x)); 0]$$
$$f3(n,x) = [len(f1(n,x)); 1 + f3(n\text{-}1,x)]$$

Since the first element in the list representing the body of $f3(n, x)$, $len(f1(n, x))$, does not contain a pattern function, we can simply use that term for the top term. For the bottom term, we must look at the final element in the list, which is $1 + f3(n\text{-}1, x)$. Since this term does contain a call to $f3$, we must introduce a new function that recurs in the same way that $f3$ recurs but does not use the entire list of proof steps in its body. Instead, this function will only represent the final term. This function is identical to $f2$, whose definition is shown below:

$$f2(0) = 0$$
$$f2(n) = 1 + f2(n\text{-}1)$$

In order to generate this function, we actually first generate a similar function with an extra variable x, as in $f3$. However, since this x is redundant we eliminate it and arrive at the function above.

In order to prove $len(f1(n, x)) = f2(n)$, we induct on n. The base case is $len(f1(0, x)) = f2(0)$, which is proved by simplification. The induction step is $len(f1(n, x)) = f2(n) \rightarrow len(f1(1 + n, x)) = f2(1 + n)$, which is also proved by simplification.

For the function $f4(n, x)$, we obtain the top term $len1(f1(n, x), f2(n\text{-}n))$ and the bottom term $f2(n)$. The top term is easily extracted from the body of $f4'$ after expanding the non-recursive function $f4$. The bottom term is obtained by simplifying the function $f4'$ after expanding $f4$, using only the final term in the list. This would produce the function $f4''$, shown below. However, since the function $f4''(n, k, x)$ is tail recursive and does not modify any accumulator, it can be simplified to $f2(k)$. Recalling that $f4'$ was originally an expansion of $f4$, which bound k to n, this gives us $f2(n)$.

$$f4(n,x) = f4'(n,n,x)$$
$$f4'(0,k,x) = [len1(f1(n,x), f2(k\text{-}n)); f2(k)]$$
$$f4'(n,k,x) = [len1(f1(n,x), f2(k\text{-}n)); f4'(n\text{-}1,k,x)]$$

$$f4''(0,k,x) = f2(k)$$
$$f4''(n,k,x) = f4''(n\text{-}1,k,x)$$

In order to prove the top term and the bottom term equal, we must keep track of the variables that were linked to n and those that were linked to k. We separate the two by adding a new variable m with the constraint that $m \leq n$. This gives the theorem $len1(f1(m,x), f2(n\text{-}m)) = f2(n)$. This is proved by induction on m. The base case is $len1(f1(0,x), f2(n\text{-}0)) = f2(n)$, proved by simplification, and the induction step is $len1(f1(m,x), f2(n\text{-}m)) = f2(n) \rightarrow len1(f1(1+m,x), f2(n\text{-}(1+m))) = f2(n)$, also by simplification. By instantiating this theorem with $m = n$, we arrive at our goal for the right hand side of $len1(f1(n,x), f2(n\text{-}n)) = f2(n)$. Notice that the lemmas for the right and left hand sides both have $f2(n)$ as their right hand side. Thus we can use these two equalities to prove $len(f1(n,x)) = len1(f1(n,x), f2(n\text{-}n))$. Instantiating this theorem with $n = length(x)$ and then attempting to equate the result with our original goal, $len(x) = len1(x,0)$ drives us to prove the lemma $f1(length(x), x) = x$. With this lemma, we are able to equate our theorem to the original goal. In this final step, we differentiate between *len* and *length* because *len* is an object level function that is derived from the input theorem, and *length* is an underlying part of the proof process that is used during every proof, and only by coincidence are the two similar functions used in this proof. In Section 2.2 we describe how we are able to discover the predicate $n = length(x)$. The substitution of x for $f1(length(x), x)$ is a step that commonly occurs during proof

about lists, as it corresponds to removing the expansion of x that was introduced during the generation of the instance proof.

2.10 Representing the Proof

Before we delve into the details of how we find patterns in proofs, we need to describe our representation for proofs in more detail.

The datatype we will be using to present our algorithm is below.

$$hybr ::= symbol$$

$$hybr ::= integer$$

$$hybr ::= list\langle hybr\rangle \text{ ; list must be non-empty}$$

$$hybr ::= HCall(symbol, list\langle hybr\rangle)$$

$$hybr ::= PCall(symbol, natural, list\langle hybr\rangle)$$

This is the same datatype we were using in the previous section, except here we have separated out the hybrid calls (HCalls) and the pattern function calls (PCalls). Note that pattern calls always contain an integer because the first argument to any pattern function we generate is always n. The main reason we have introduced this datatype instead of just using nested lists as before is to make it clear how we think about the proof terms we are manipulating. In reality, our implementation just uses nested lists and we differentiate between hybrid calls and pattern calls using a naming convention that assumes functions of the form F?? are pattern functions. The type descriptor above is presented using ML like notation, as is much of the pseudocode below. The hybr type is either a symbol or an integer, or it is built up by composing smaller hybr types. The third says that a hybr type can be made from a list of hybr types, the next says that it can be an HCall of a symbol (representing the function name) and a list of hybrs (representing the arguments of the call). The

final constructor, PCall, is similar to HCall, except that we list one argument to the function separately since we know all pattern functions have an integer argument.

2.11 Detecting Patterns

Earlier we presented an example that showed a sequence of steps that replaced parts of a hybrid proof with calls to functions that generate equivalent proof terms. We call such functions *pattern functions* because they are introduced when some repeating pattern is found in the proof. Below we present pseudocode for the algorithm for finding such patterns. The code below will be called repeatedly, each time replacing some set of subterms with calls of pattern functions, until no more patterns are found.

The function $find\text{-}pattern$ is the top level function that will be called repeatedly until it returns false. The variable p is the proof we are currently processing and the variable *path* tells us which subterm we are currently looking at. When the process is complete, it will have transformed a proof by introducing calls of a pattern functions if successful. The pseudocode below is presented using ML style type specifiers. For example, the formal argument $p : list\langle hybr\rangle$ means that the argument p is of type $list\langle hybr\rangle$. The type $list\langle hybr\rangle$ is a parameterized list type, meaning it is a list that contains only *hybr* objects.

We will be using the following definitions. A path is a sequence of naturals. We use paths to select subterms of a term. To use a path to select a term, we take the first natural in the sequence and use it to select a subterm of the original term. Then, we repeat the process on the subterm with the remainder of the sequence. We call the subterm selected in this manner usepath(p, x), where p is the path and x is the original term. We also use the function usepath(p, x) in a similar manner to

select portions of hybrid terms. If we have a subterm located at path p, all subterms located at paths which are prefixes of p we call *ancestors*. The *highest* ancestor of a set of ancestors is the one located at the shortest prefix.

The function *find-pattern* works from the bottom up, first making recursive calls on all subterms. Note that this differs from the examples we have given in previous sections, which were presented as top-down algorithms. We presented examples earlier using a top-down technique because it is somewhat simpler. However, for efficiency it is better to find patterns bottom-up. This is because choosing the wrong pattern in a top-down manner may cause a lot of backtracking, whereas when we find a pattern using a bottom-up search there will not need to be any backtracking because the patterns it depends on have already been found.

When we process a given subterm, we skip over any subterms that have had their children modified during this iteration. This lessens the chance that ancestors will have children that have not yet been processed, making finding a pattern unlikely. To process a term, we first locate each sequence of ancestors with the following properties: they have the same *heading* as the current term; they are aligned such that given the path from the top of the proof to a term other than the last in the sequence, appending a fixed step to that path gives the path from the top of the proof to the next term; and finally, the sequence ends at our current subterm. *Headings* are simple identifiers that can be used to see quickly if two terms could possibly be of the same form. The cases for *find-pattern* on function calls and lists are similar, while *find-pattern* terminates on symbols and integers. For function calls, we use the function name as a heading. For lists, we use a list containing the headings of its elements. For each sequence we find, we call the function *check-sequence* to see if it can be represented with a pattern function. If we are successful, we modify

```
find-pattern(p : list⟨hybr⟩, path) =
    for each element of p with index idx
        tmp := find-pattern(p, idx :: path)
        modified := modified|tmp
    if modified
        return true
    else
        find every sequence of ancestors with same
        heading as usepath(path, p) which have
        a fixed path step separating them and which
        terminates at path
    for each sequence represented as (highest, step, len) found:
        (success, pcall) :=
            check-sequence(p, highest, step, len)
        if success
            replace subterm of p located at path highest
            with pcall
            return true
    return false
```

Figure 2.1: The function find-pattern

the proof p by replacing the highest ancestor with a call to the given pattern function. Note that from a technical standpoint, replacing the highest term may make the current path invalid for the modified proof. Although our implementation deals with this problem, we will ignore those details during presentation of this code.

The function *check-sequence* takes a sequence of nested subterms and tries to find a pattern function that can be called to replace each of them. A sequence is represented by three variables: *top*, *step*, and *len*. The variable *top* is a path that gives the first term in the sequence. The variable *step* is a path that, when appended to the path for the previous term, gives the next term. The variable *len* tells how many terms are in this sequence. To find a pattern function to represent this sequence, *check-sequence* first forms a list of truncated terms by replacing the subterm in position step of each term in the sequence with a unique constant *REC*.

This removes redundant information since, for all terms except the last, the next term in the sequence is located at that position. Each pair of adjacent terms in the sequence is passed, in order, to a function *find-delta-pattern*, defined below, along with an index that represents which value of n the first item of the pair will correspond to in the final pattern function call. The highest term in the sequence corresponds to $n = len$ and the final term corresponds to $n = 1$. The term one step beyond the final term is considered the base case and corresponds to $n = 0$. The function *find-delta-pattern* will return any patterns recognized in these pairs of terms. A pattern is represented by a term that may contain pattern functions, although the top level term is generally not a pattern function call. A pattern may also contain the special symbol n. To test a pattern on a given term, the pattern is evaluated with n set to the actual value of n corresponding to that term. If all terms match the pattern, a pattern function is created by *create-fun*. The function *create-fun* takes a term that may contain the variable n and uses this term to create a function with this term as its body. This function recurs with n set to n-1 at the position in the body specified by the path *step*. For the base case, the term obtained by appending the path *step len* times to the path of the top term will be used. The final argument to *create-fun* specifies the value that n will be bound to in the call to the function that was created, and *create-fun* returns that call. During this process, *create-fun* may also extract constants from the body and do some simplification of the function if necessary.

The function *find-delta-pattern* takes a pair of terms along with an integer n representing where in the sequence the first element of the pair is located and tries to return a pattern that fits both elements in a uniform way. Stated another way, *find-delta-pattern* returns a hybrid term *pat* with free variable n such that

check-sequence$(p, top, step, len)$ =
    for each pair (x, y) of adjacent subterms
        of the sequence represented by (top, step, len)
        where x is i number of steps from *top*
        let x' and y' be x and y, where we have replaced
        the subterm at position *step* within both x and y
        with the new symbol 'REC
        call *find-delta-pattern*$(x', y', (len\text{-}i))$
        and collect the results in the list l
    for every pattern pn occurring more than once in list l
        test pn on all subterms
        by plugging in the appropriate value of n
        for a given subterm and evaluating pn
        then testing for equality
        if successful
            $pcall :=$ *create-fun*$(pn, step, len)$
            return $(true, pcall)$
    return $(false, nil)$

Figure 2.2: The function check-sequence

binding n to $(len\text{-}i)$ gives the $i + 1^{th}$ element of our sequence, which is 1-based. In the cases below, the variable idx is used to represent which value of n the variable x corresponds to. Since n decreases as we move through the sequence from the highest to the lowest term, y corresponds to the case where n is idx-1. The simplest case is for two symbols. They must match exactly or else no pattern is returned. Please note, each of these cases has a mirrored case where x and y have been swapped, which we have not shown. These mirrored cases may differ slightly from the original cases to compensate for the reversed ordering of the pair. Please note that these cases are presented in reverse order of priority, such that the last one will be applied first if applicable. Mirrored cases have the same priority as the original cases. For example, a pair that consists of a symbol and then an HCall would first be attempted by the mirrored *find-delta-pattern* where x is any hybrid and y is an HCall, before

it was attempted by the *find-delta-pattern* where x is a symbol and y is any hybrid.

$$
\begin{aligned}
&\textit{find-delta-pattern}(x : \textit{symbol}, y, \textit{idx}) = \\
&\quad \text{if } x = y \text{ then} \\
&\qquad \text{return } x \\
&\quad \text{else} \\
&\qquad \text{return failure}
\end{aligned}
$$

Two integers are fit to an expression of the form $a * n + b$. Note that if the expression below is evaluated with n set to idx it yields x, and if it is evaluated with n set to idx-1 it yields y. In the expression below, the variable n is a formal variable while x, y, and idx all have actual values that will be used to compute the constant coefficients for the expression.

$$
\begin{aligned}
&\textit{find-delta-pattern}(x : \textit{integer}, y : \textit{integer}, \textit{idx}) = \\
&\quad \text{return } \textit{HCall}(+, [x\text{-}((x\text{-}y) * \textit{idx}); \\
&\qquad\qquad \textit{HCall}(*, [x\text{-}y; n])])
\end{aligned}
$$

If *find-delta-pattern* is called with two hybrid calls of the same function, their arguments are pairwise recursively delta'd to form the arguments to that function. If an *HCall* is delta'd with a different *HCall* or an integer or symbol, the function *find-subterm-pattern*, defined below, is used. The function *map*2 below takes a binary function along with two lists of arguments and returns a list formed by calling that function pairwise on elements from each list. As an example, say that x was $foo(1)$ and y was $foo(2)$. For this case, the top function symbols are the same, so *find-delta-pattern* will recur by comparing the arguments. In this case, both are integers so the *find-delta-pattern* above for two integers will be used.

$find\text{-}delta\text{-}pattern(x : HCall(fx, xargs), y, idx) =$

if y matches $HCall(fy, yargs)$ and $fx = fy$ then

$xyargs := map2((fn(a, b) \rightarrow$

$find\text{-}delta\text{-}pattern(a, b, idx)),$

$xargs,$

$yargs)$

if successful

return $HCall(fx, xyargs)$

$find\text{-}subterm\text{-}pattern(x, y, idx)$

The function *find-subterm-pattern* checks to see if either term is a subterm of the other. Recall that this function is used after *check-sequence* has already created a sequence of side terms by replacing the term at step *path* with a new symbol 'REC. Furthermore, we may also have already called another case of *find-delta-pattern*, for example the case above which compares to HCalls with the same top function symbol by comparing their arguments. More specifically, if the sequence created by *check-sequence* was:

$$foo('REC, cdr(cdr(x)))$$
$$foo('REC, cdr(x))$$
$$foo('REC, x)$$

Then the case above for *find-delta-pattern* where x and y are both HCalls with the same top function symbol would have been used. Since this case recurs by calling *find-delta-pattern* on its arguments, we could then arrive at a call to *find-delta-pattern* where x is $cdr(cdr(x))$ and y is $cdr(x)$. This is the case that the code below is meant to handle.

The heuristic for this case is to return a call of a pattern function that repeats the shell of one term at the position where the other was found in that term. For example, if one term is $cdr(x)$ and the other is x then we would discover the pattern of nested cdrs. Depending on the size of the subterm, it may already contain several repetitions of the pattern. For example, $cdr(cdr(x))$ could be found inside of $cdr(cdr(cdr(x)))$. This generates a pattern of repeating cdrs. However, the call to the pattern function must represent the fact that there are two and three, respectively, repetitions of this function. Also, the call returned must indicate where within the sequence the pattern reaches a given number of repetitions. If, for example, the sequence consists of three terms, where the first is a repetition of three and the second is a repetition of two, the general pattern is that the n^{th} term has n repetitions. If, instead, the sequence consists of three terms where the first is a repetition of four and the second is the repetition of three, the general pattern is that the n^{th} term has $n + 1$ repetitions. This would mean that the adjustment, below, would be 0 and 1 respectively.

$find\text{-}subterm\text{-}pattern(x : HCall(fx, xargs), y, idx) =$

if y is a subterm of x at path p

replace the subterm at path p in x with new symbol 'REC

Let $pcall := create\text{-}fun(x, p, HCall(+, [adj; n]))$

for suitable natural number adj, an adjustment based on idx and x

such that x is equal to the evaluation of $pcall$

with n set to idx

Test the pattern $pcall$ on y

with n set to idx-1

if successful

return pcall

return failure

If $find\text{-}delta\text{-}pattern$ is called with two pattern calls, they are recursively delta'd in much the same way as two HCalls. If a $PCall$ is delta'd with another term y, we first fully expand any PCalls in y to remove any that may be present, then we match our $PCall$ against y by repeatedly unfolding its definition to see if y can be represented by some call to the same pattern function.

$$
\begin{aligned}
&\textit{find-delta-pattern}(x : PCall(fx, nx, xargs), y, idx) = \\
&\quad \text{if } y \text{ matches } PCall(fy, ny, yargs) \text{ then} \\
&\qquad \text{if } fx = fy \text{ then} \\
&\qquad\quad xyargs := map2((fn(a, b) \rightarrow \\
&\qquad\qquad \textit{find-delta-pattern}(a, b, idx)), \\
&\qquad\qquad\quad xargs, \\
&\qquad\qquad\quad yargs) \\
&\qquad\quad \text{return } PCall(fx, \\
&\qquad\qquad \textit{find-delta-pattern}(nx, ny, idx), xyargs) \\
&\qquad \text{else} \\
&\qquad\quad \text{return failure} \\
&\quad \text{else} \\
&\qquad \textit{pattern-match}(x, eval(y))
\end{aligned}
$$

2.12 Creating the Proof

In the previous sections, we showed how to generate a hybrid term to generate our proof using calls of hybrid functions. In this section, we show how to translate hybrid terms into traditional proofs. We use the function *hybr-to-trad* to do this. This function is presented in four pieces, according to the datatype *hybr*. A symbol is transformed into a trivial proof that it is equal to itself:

$$\textit{hybr-to-trad}(x : \textit{symbol}) = [x; x]$$

A list of hybrid terms is converted into a traditional proof by first recursively converting all the elements of the list, and then appending all sublists into one flat list of traditional terms:

$$\textit{hybr-to-trad}(x : list\langle hybr\rangle) =$$
$$flatten(map(\textit{hybr-to-trad}, x))$$

To convert a hybrid call into a traditional proof, we first convert all of the arguments of the call into traditional proofs. For each such proof, we generate a lemma that equates the first and last terms of that proof. To do this, we use procedure *iter*, which applies the function *make-lemma* to all elements of l, ignoring any return values. We then return the sequence of two traditional calls (TCalls) that are formed by taking the first term in each proof for the corresponding argument and the last term in each proof for the corresponding argument:

$$\begin{aligned}
&\textit{hybr-to-trad}(HCall(fname, args)) = \\
&\quad l := map(\textit{hybr-to-trad}, args) \\
&\quad iter(\textit{make-lemma}, l) \\
&\quad tl := map(top, l) \\
&\quad bl := map(bot, l) \\
&\quad \text{return } [TCall(fname, tl); TCall(fname, bl)]
\end{aligned}$$

To translate a pattern function call to a traditional proof, we first replace the value of n in the call with a new variable n'. Then we expand the call once and replace the recursive call in the body with a traditional proof that represents the induction hypothesis. Once this is done, we can call *hybr-to-trad-ih* on the body. This function is identical to *hybr-to-trad* except that it does not prove the induction hypothesis and instead assumes it as a hypothesis to the overall theorem. We similarly call *hybr-to-trad* on the base. Each of these is made into a lemma that is used to prove the main theorem by induction. Once this theorem has been proved, the instance of the theorem that corresponds to the original call is returned.

As an example, recall the function $f3$ from Section 2.9:

$$f3(0, x) = [len(f1(0, x)); 0]$$
$$f3(n, x) = [len(f1(n, x)); 1 + f3(n\text{-}1, x)]$$

Expanding the call $f3(n', x)$ once gives the body $[len(f1(n', x)); 1 + f3(n'\text{-}1, x)]$. Recall from Section 2.9 the top term of this call is $len(f1(n', x))$ and the bottom term is $f2(n', x)$. If we replace the recursive call in the body of $f3$ with the list containing the top and bottom term with n'-1 substituted for n', we get: $[len(f1(n', x)); 1 + [len(f1(n'\text{-}1, x)); f2(n'\text{-}1, x)]$. Calling *hybr-to-trad-ih* on this proof term will first prove the lemma $len(f1(n', x)) = 1 + len(f1(n'\text{-}1, x))$. With a normal call to *hybr-to-trad* we would then prove $len(f1(n'\text{-}1, x)) = f2(n'\text{-}1, x)$ but with *hybr-to-trad-ih* we instead skip that step and add it as a hypothesis to the combined lemma which equates the top and bottom terms. This lemma is:

$$len(f1(n'\text{-}1, x)) = f2(n'\text{-}1) \rightarrow len(f1(n', x)) = 1 + f2(n'\text{-}1)$$

After proving the base case $len(f1(0, x)) = 0$, we combine these two lemmas to prove the final goal by induction: $len(f1(n', x)) = f2(n')$. In order to make sure these lemmas are used correctly, we carefully guide ACL2 by setting the current theory to be minimal and enabling only these lemmas.

To complete the example from Section 2.9, we would form a lemma for the function f4(n,x) that equates the top and bottom terms for this function in the same way we did for f3(n,x). This gives $len1(f1(n, x), f2(n\text{-}n)) = f2(n)$. Now, we combine the two lemmas by equating the equality of top terms with the equality of bottom terms. Simplifying this lemma using the fact that the bottom terms are equal gives that the equality of the top terms is true. We complete the proof

by substituting $length(x)$ for n and equating the result with the original theorem, using any mismatches to suggest lemmas. This final step is performed only once, at the top level. We know to substitute $length(x)$ because it is the generalization of the condition we used to generate the original finite case. Section 2.2 describes the process of finding an generalizing this case.

The above paragraph is a slight oversimplification of the actual process. The reason for this is that the condition $n = length(x)$ which is generated in Section 2.2 is actually a predicate on n and x rather than an equality. Therefore, the reality is that instead of substituting $length(x)$ for n and then simplifying, we actually add this predicate as a hypothesis and then simplify. This gives us a proof of our original theorem with this added predicate. If we had used an equality instead of this predicate, we could then eliminate the extra hypothesis by simply substituting $length(x) for n$ in the hypothesis, making it trivially true. Instead, we use the original theorem with the added predicate to prove the base case of an induction on the original theorem using the scheme $ifun$ generated in Section 2.2. The induction step for this scheme is trivial, and the base case corresponds to the original theorem with the added predicate.

$hybr\text{-}to\text{-}trad(call : PCall(fname, n, args)) =$

$call' := PCall(fname, n', args)$ where n' is

a new variable

$body :=$ expand $call'$ one step using definition of $fname$

$ih := [top(call); bot(call)]\{n' \leftarrow n'\text{-}1\}$

$body' :=$ replace the recursive call in $body$ with ih

$step := hybr\text{-}to\text{-}trad\text{-}ih(body', ih)$

$make\text{-}lemma(step)$

$base :=$ expand $call'$ using $fname$'s base case

$zero := hybr\text{-}to\text{-}trad(base)$

$make\text{-}lemma(zero)$

$make\text{-}lemma\text{-}ind(top(call) = bot(call))$

by induction on n' by -1

return $[top(call)\{n' \leftarrow n\}; bot(call)\{n' \leftarrow n\}]$

One question that arises during this step is: have we introduced circular reasoning, by trying to break the original theorem that required some kind of sophisticated generalization or lemmas to complete induction down into lemmas that may themselves be difficult to prove by induction? Fortunately, this is not the case. Because these lemmas are created from pattern functions, which all follow the same simple type of recursion, these lemmas can be proved automatically without generalization or additional lemmas using an inductive theorem prover such as ACL2.

2.13 Results

We have implemented this system in ACL2, and have tested it using the theorems shown in Table 2.1. Results for our system appear in the column labeled GPI,

while results for the unmodified ACL2 theorem prover appear in the column labeled ACL2. The DC column contains the results for a related system, the Divergence Critic, which we discuss below. The Sec column contains the time in seconds for GPI to prove the given theorem, including time for ACL2 to check the proof. Our system generated ACL2 proofs for the checked theorems in the GPI column. These proofs were generated by sending ACL2 a sequence of definitions and lemmas leading up to the final theorem. More than 75% of these theorems cannot be proven automatically by ACL2. Our system succeeds on all but one of these theorems (31). The primary reason for this failure is that our system cannot currently recognize the interaction of multiple variables beyond addition. We someday hope to explore adding such functionality. Please note, we use our own definitions for Peano arithmetic, rather than the built-in ACL2 arithmetic definitions. This allows us to focus on our heuristics in the context of simpler theorems. The results in the ACL2 column are also using our own definitions. If we attempted to prove these theorems with ACL2 using using its own definitions and arithmetic libraries, it would most likely prove many more of them.

Our list of theorems was taken from a paper on the Divergence Critic (DC) [22]. While these are largely positive examples of theorems that the divergence critic can prove, our system was able to prove theorems 18, 32, 33, and 34, while DC was not. We also added three new theorems: 21, 22 and 23. Unfortunately, there was no implementation available for the Divergence Critic, so we were unable to test new theorems. The reason the Divergence Critic was unable to prove theorems 18 and 34 was that the generalization required uses a function, append, that is nowhere present in the theorem. This is a problem that many systems have when trying to generalize. Our system does not suffer from this problem because it looks for

patterns in proofs rather than trying to manipulate the goal formulas directly. For theorems 32 and 33, our system was able to solve these problems because the nested inductions resulted in patterns that did not increase greatly in complexity. With other approaches, each new induction introduces a great deal of choices (for which variable to induct, which substitutions, how to generalize), which makes multiple inductions very difficult. Although the results for our system are very similar to the Divergence Critic, the two systems are quite different in their manner of operation. Our system is able to use patterns to introduce and take advantage of new recursive definitions.

2.14 Summary

We have presented a novel technique that shows how proof instances can be exploited to prove theorems that are not automatically provable by ACL2 alone. Our technique generates proofs for instances of the goal theorem. These proofs are represented using a hybrid proof term form that allows us to detect patterns in these proofs more easily. We use these patterns to generate a sequence of lemmas that lead to a first order inductive proof of the goal theorem that is automatically checked by ACL2. This technique is not only useful, but it also differs significantly from other current techniques for proving inductive theorems.

Table 2.1: Summary of theorems proved

No.	Theorem	ACL2	GPI	DC	Sec
1	x+s(x)=s(x+x)	×	√	√	6.4
2	dbl(x)=x+x	×	√	√	5.4
3	len(ap(x, y))=len(ap(y, x))	√	√	√	271.0
4	len(ap(x, y))=len(x)+len(y)	√	√	√	51.4
5	len(ap(x, x))=dbl(len(x))	×	√	√	123.7
6	even(x+x)	×	√	√	5.3
7	odd(s(x)+x)	×	√	√	23.4
8	$even_m(x + x)$	×	√	√	5.2
9	$odd_m(s(x) + x)$	×	√	√	5.2
10	half(x+x)=x	×	√	√	10.8
11	half(s(x)+x)=x	×	√	√	8.7
12	len(rev(x))=len(x)	√	√	√	25.1
13	rv(rv(x))=x	√	√	√	49.5
14	rv(ap(rv(x), [y]))=y::x	×	√	√	58.7
15	rv(ap(rv(x), [y]))=y::rv(rv(x))	√	√	√	395.6
16	len(rv1(x, nil))=len(x)	×	√	√	11.8
17	rv1(x, y)=ap((rv x), y)	√	√	√	26.5
18	rv1(rv1(x, nil), nil)=x	×	√	×	16.4
19	rv(rv1(x, nil))=x	×	√	√	32.5
20	rv1(rv(x), nil)=x	×	√	√	125.0
21	len1(x, 0)=len(x)	×	√	?	64.1
22	rv1(x, nil)=rv(x)	×	√	?	125.4
23	ap(ap(x, x), x)=ap(x, ap(x, x))	×	√	?	321.0
24	rot(len(x), x)=x	×	√	√	225.4
25	len(rot(len(x), x))=len(x)	×	√	√	226.5
26	rot(len(x), ap(x, y::[]))=y::rot(len(x), x)	×	√	√	673.5
27	len(rv1(x, y))=len(x)+len(y)	×	√	√	219.0
28	nth(x, nth(y, z))=nth(y, nth(x, z))	×	√	√	96.6
29	nth(x, nth(y, nth(z, w)))= nth(z, nth(y, nth(x, w)))	×	√	√	192.6
30	len(rv(ap(x, y))) = len(x) + len(y)	×	√	√	753.2
31	s(x) * y = y + (x * y)	√	×	√	602.5
32	x * y = y * x	×	√	×	569.6
33	x + y + z + w = z + y + x + w	×	√	×	375.5
34	rv1(rv1(x, y::nil), z) = y::rv1(rv1(x, nil), z)	×	√	×	62.4

Chapter 3

Backtracking in ACL2

In the previous chapter, we introduced a technique that looked at proofs for small examples and tried to generalize them using pattern matching. In this chapter, we introduce another technique for finding lemmas and generalizations. This technique uses a more traditional approach, trying to prove the goal theorem directly and using failures to suggest new lemmas. Both techniques extend the range of theorems ACL2[15] can prove, and since there is probably no one technique that can address all relevant theorems, it can be useful to have different techniques available.

Often, ACL2 has to choose between several promising alternatives during the course of a proof. For example, a given theorem may suggest three possible induction schemes. ACL2 will choose one and proceed. However, if the proof fails, ACL2 has no mechanism for returning to the point at which the choice was made and attempting an alternate induction. In this paper, we describe an extension to ACL2 that allows such backtracking to occur. This extension enables us to experiment with many new theorem proving heuristics.

We describe two such heuristics and show how they can be used to automat-

ically prove theorems that ACL2 could not prove automatically before. The first of these is an induction variable matching algorithm that allows ACL2 to automatically generate new induction schemes for theorems about functions with unmeasured variables. This algorithm in based on a paper by Kapur and Subramaniam [14] and allows ACL2 to prove theorems such as `(equal (rot (len x) (append x y)) (append y x))`, where an induction scheme must be discovered that substitutes `(append y (list (car x)))` for `y`, when no such scheme is suggested by the functions involved.

The second heuristic is related to cross fertilization. There are times when ACL2 will choose to cross fertilize and generalize when it would be better to skip cross fertilization and generalize directly. With our extension, both possibilities can be tried.

Although our first heuristic is based on a paper by Kapur and Subramaniam, we have made several notable contributions beyond what was outlined in the paper. First, there is no existing implementation for the algorithms described in the paper. Ours is the only implementation currently available for any system. Second, we adapt their heuristics to the ACL2 system. This involves, among other things, dealing with ACL2's destructor style recursion and induction, and dealing with theorems that are not strict equalities, but they may contain hypotheses. Finally, we have integrated this heuristic with the second heuristic described in the paragraph above. As shown later in this section, the two can work together to prove even more complex theorems.

3.1 Induction Variable Matching

In ACL2, when an induction scheme contains unmeasured variables, the scheme can be modified to yield any substitution for those variables. Since only unmeasured variables are modified, the measure will be unchanged, and the scheme will remain sound.

Although this allows a great deal of flexibility in choosing an induction scheme, it can be difficult to find the right substitutions for a given variable. The technique we use is based on a paper by Kapur and Subramaniam [14]. The main idea is to replace the unmeasured variables with new constrained functions in the induction hypothesis, and then attempt to find definitions for them by attempting to match the induction conclusion ("IC") and the induction hypothesis ("IH") after simplification. Differences are eliminated by removing recursive functions using case splits. After definitions are found for the constrained functions, they are substituted back into the original induction step. If any differences remain, lemmas are speculated to remove them.

For an example of when induction variable matching can be helpful, consider the theorem[1]

```
(implies (and (true-listp x) (true-listp y))
         (equal (rot (len x) (append x y))
                (append y x)))
```

where `rot` is defined as:

```
(defun rot (n x)
```

[1]Here we show the full theorem with true-listp hypotheses. In the rest of the paper, we will present this example as if these hypotheses were not needed. However, our system must track these in order to correctly prove this theorem and does so.

```
(if (zp n)
    x
  (rot (1- n) (append (cdr x) (list (car x))))))
```

ACL2 will attempt to induct on `(cdr x)`, leaving `y` unchanged. This gives

```
(equal
 (rot (len (cdr x)) (append (cdr x) y))
 (append y (cdr x)))
```

for the induction hypothesis. After stepping and simplifying the induction conclusion, we get:

```
(equal
 (rot (len (cdr x))
      (append (append (cdr x) y) (list (car x))))
 (append y x))
```

The IH,

```
(equal
 (rot (len (cdr x)) (append (cdr x) y))
 (append y (cdr x)))
```

cannot be applied. But note that if in this IH we further replaced y with `(append y (list (car x)))` and we knew that append was associative, the new IH would apply and finish the proof. The challenge is finding this instantiation of `y`.

To do this, we start by replacing `y` with the constrained function `(F x y)` in the induction hypothesis to get the following induction hypothesis:

```
(equal
 (rot (len (cdr x)) (append (cdr x) (F x y)))
 (append (F x y) (cdr x)))
```

Attempting to match the LHS of the IH and simplified IC reveals the following differences:

```
IH (append (cdr x) (F x y))
IC (append (append (cdr x) y) (list (car x)))
```

We try to match these by assuming the base case for the inner most recursive call in the IC and simplifying. In this case, the inner most call is `(append (cdr x) y)`. Since the base case for append is `(endp x)` and `(cdr x)` is in the position for x, we substitute `(cdr x)` for `x` within `(endp x)` and get `(endp (cdr x))`. Simplifying these two terms under that hypothesis gives us:

```
IH (F x y)
IC (append y (list (car x)))
```

which gives us a definition for `F`. Since this definition was generated using a special case, we substitute it back into the constrained IH,

```
(equal
 (rot (len (cdr x))
      (append (cdr x) (append y (list (car x)))))
 (append (append y (list (car x))) (cdr x)))
```

and then compare this to the simplified IC,

```
(equal
 (rot (len (cdr x))
```

```
      (append (append (cdr x) y) (list (car x))))
 (append y x))
```

to see what differences remain.

On the LHS, this gives us:

```
IH (append (cdr x) (append y (list (car x))))
IC (append (append (cdr x) y) (list (car x)))
```

On the LHS, this difference can be proved as a lemma. We orient the equation by putting the biggest term[2] on the lhs. We break ties by considering terms with conses farther to the left as heavier. If we also generalize by replacing common subterms with new variables, we obtain the associativity of `append`: `(equal (append (append x y) z) (append x (append y z)))`

On the RHS, we have these differences:

```
IH (append (append y (list (car x))) (cdr x))
IC (append y x)
```

If we equate the IH and IC, the resulting lemma is a special case of associativity, and can be recognized as redundant by proving it via simplification from the earlier lemma.

3.2 Multiple Generalizations

It is well known that many theorems must be generalized before they can be proved. For example, `(equal (rev1 x nil) (rv x))` is typically generalized to `(equal (rev1 x a) (append (rv x) a))`. Although ACL2 already has the capability to

[2] as measured by acl2-count

generalize theorems, often it will choose a bad generalization. If this happens, ACL2 will not try another generalization; it will simply fail. Our extension allows alternate generalizations to be attempted.

The system generates two alternatives during every induction. First, it will try to cross fertilize and then generalize any remaining goals before another induction. If that fails, it will throw away the IH and generalize the remaining goals without cross fertilization. Below we give several examples of when this alternate generalization will succeed.

3.2.1 Reverse Example

As an example, consider the theorem `(equal (rv1 x nil) (rv x))`, where rv and rv1 are defined as:

```
(defun rv (x)
  (if (endp x)
      nil
    (append (rv (cdr x)) (list (car x)))))


(defun rv1 (x a)
  (if (endp x)
      a
    (rv1 (cdr x) (cons (car x) a))))
```

ACL2 will induct on `(cdr x)` to prove this theorem. After simplification and destructor elimination, the induction step will be:

```
(implies (equal (rv1 x2 nil) (rv x2))
         (equal (rv1 x2 (list x1))
```

```
        (append (rv x2) (list x1))))
```

ACL2's normal behavior is to cross fertilize after this step, yielding:

```
(equal (rv1 x2 (list x1))
       (append (rv1 x2 nil) (list x1)))
```

Cross fertilization has reintroduced the constant `nil` into the accumulator of `rv1`. This will make proving the above goal difficult. If instead, we throw away the IH and generalize by replacing `(list x1)` with `x3`, we get:

```
(equal (rv1 x2 x3)
       (append (rv x2) x3))
```

which can be proved by ACL2.

3.2.2 Rotate Example

Consider the theorem `(equal (rot (len x) x) x)`. After simplification and destructor elimination, the induction step will be:

```
(implies
 (equal (rot (len x2) x2) x2)
 (equal (rot (len x2) (append x2 (list x1)))
        (cons x1 x2)))
```

After cross fertilization, we get:

```
 (equal (rot (len x2) (append x2 (list x1)))
        (cons x1 (rot (len x2) x2)))
```

which ACL2 further generalizes to the non-theorem:

```
(defthm car-ap-cons
  (equal (car (append (cons a b) c))
         a))

(defthm cdr-ap-cons
  (equal (cdr (append (cons a nil) c))
         c))

(defthm append-cons
  (consp (binary-append (cons x3 nil) z))
  :rule-classes :type-prescription)

(defthm cons-ap
  (implies (syntaxp (not (equal x ''nil)))
           (equal (cons a x)
                  (append (cons a nil) x))))
```

Figure 3.1: cons to append normalization rules

```
(implies (and (integerp i)
              (<= 0 i))
         (equal (rot i (append x2 (list x1)))
                (cons x1 (rot i x2))))
```

If instead, we skip cross fertilization and throw away the induction hypothesis, we get:

```
 (equal (rot (len x2) (append x2 (list x1)))
        (cons x1 x2))
```

In this case, there are no common subterms across the equality, so generalization fails. However, notice that the element `x1` occurs at the end of the list in the accumulator on the LHS and at the beginning of the list on the RHS. If we use the rules in Figure 3.1 to normalize lists, the goal above becomes:

```
(equal (rot (len x2) (append x2 (list x1)))
       (append (list x1) x2))
```

Now we can generalize, because the term `(list x1)` appears on both sides. This gives

```
(equal (rot (len x2) (append x2 x3))
       (append x3 x2))
```

This theorem we proved earlier using unmeasured variable matching.

We developed the rules in Figure 3.1 because we felt that ACL2s default way of normalizing lists was incorrect. Normalization should provide a single form for a given value whenever possible, but as the example above shows, ACL2 represents adding an element on the front of a list with cons, while it represents adding an element on the end of a list with an append. These rules correct that problem, allowing generalization to act naturally. Furthermore, they are designed so that they can be left enabled and will not interfere with other proofs.

3.3 Implementation

Our implementation uses ACL2's simplification and generalization routines along with our own version of induction. Our induction routines replace unmeasured variables in the induction hypothesis with constrained functions for which we will later find definitions. Instead of using ACL2's top level prover, we have our own control flow that allows induction to be entered and exited recursively. Below we present pseudocode for our implementation. The top level function, shown in Figure 3.2 below, is called *bprove*. It takes a term and attempts to prove it, returning either SUCCESS or FAILURE. The correctness of this prover need not be trusted, because

```
bool bprove(term x)
{
  l := bash(x)

  for each permutation p of l
    success := prove-perm(p)
    if success
      return SUCCESS
    else
      continue

  return FAILURE
}
```

Figure 3.2: The top level function for the backtracking prover.

we generate events for ACL2 to check the final proof in the end. This part of our presentation below has been omitted for simplicity.

Our prover starts with a call into the simplifier using the bash book developed by Matt Kaufmann. This simplifier returns a list of clauses. We must prove all clauses in order to prove our goal theorem. Since there may be constrained functions in the clauses that will be bound to concrete functions as we proceed, the order in which we prove the clauses is important. This is because, while proving a clause, we may discover bindings for constrained functions in that clause. These bindings will be used to remove any instances of the same constrained function in later clauses. Furthermore, different clauses may find differing bindings for the same constrained function. Therefore, it is necessary to try to prove all permutations of a given clause list. Figure 3.3 shows the pseudocode for the function *prove-perm*, which is used to prove such a permutation. For performance reasons, our implementation does not actually compute the entire set of permutations at once. Instead, we generate one

```
// l is a list of clauses
bool prove-perm(list l)
{
  while l is non-empty
    c, l := remove-clause(l)
    if there are any constrained functions in c
      c, success, bind := remove-constraints(c)
      if !success
        return FAILURE
      l := apply-subst(bind, l)

    refuted := refute(c)
    if refuted
      return FAILURE

    success := binduct(c)
    if !success
      c := generalize(c)
      success := binduct(c)
      if !success
        return FAILURE

  return SUCCESS
}
```

Figure 3.3: Prove a list of clauses

```
(clause, bool, list) remove-constraints(clause c, list bind)
{
  for each literal l in c

    if l is of the form (not (equal lhs rhs)), attempt to match
    each side of the equality against all subterms of the other
    literals in the clause, replacing lhs[rhs] with rhs[lhs]
    wherever applicable

    if an equality successfully matches, remove it from the
    clause and call remove-constraints again on the remaining
    literals, along with any bindings acquired during the match

    if l is of the form (not l'), attempt to match l'
    against the other literals in the clause

    if the match is successful, then we have found two literals
    that are negations of each other.  We return SUCCESS along
    with the substitutions returned from match and the empty
    clause.

  if any constrained functions remain, return FAILURE else return
  SUCCESS along with the modified clause and any bindings

}
```

Figure 3.4: Remove constraints

```
// returns a list of bindings for any constrained functions if
// successful
(bool, list) match(term a, term b)
{
  if neither a nor b contain any constraints
    call bprove on ‘‘a = b’’

  if a contains no constraints
    switch a and b

  if the top symbol of a is a constrained function
    bind the constrained function to b and return SUCCESS

  if a and b have the same top symbol, decompose them and attempt
  to match corresponding subterms

  if that fails, let h be the simplifying assumptions attained by
  assuming the base case for the innermost recursive function in
  a, and return match(a’, b’), where a’ and b’ are
  simplifications of a and b under hypotheses h

}
```

Figure 3.5: Match two terms modulo constrained functions

at a time. This allows us to avoid unnecessary work if we find a proof early.

For each clause, we first remove any constraints. Any bindings acquired by removing constraints are applied to the remaining clauses. Next, we attempt to discard the clause, by generating a number of finite cases of the theorem and sending them through the simplifier. If no counterexamples were found, we induct with cross fertilization and generalization. If that fails, we throw away any induction hypothesis and generalize before attempting a second induction. These calls to *binduct* use our own induction mechanism so that we can annotate the induction hypothesis with constrained functions if there are any unmeasured induction variables. The goals generated by this induction are then fed back into our prover via the function *bprove*. The proof search terminates because it is bounded by a maximum number of nested inductions. This limit is usually set to 3 but can be set to any number.

3.3.1 Removing Constraints

Removing constraints is done using the function *remove-constraints* from Figure 3.4. In order to satisfy the constraints for a given clause, we visit all pairs of literals in the clause. For a given pair, if one literal is negated and the other is not, we attempt to match them in the sense described in the following paragraph. Also, if one is a negated equality, we will attempt to match the lhs or rhs of the equality against all subterms of the other literals. If there is at least one match, we substitute terms using the equality in a suitable direction, remove the equality from the clause, and then attempt to prove the clause without the equality.

There are two techniques we use to match two terms, as shown in Figure 3.5. First, if two terms have the same top function symbol, we will decompose the terms and attempt to match their subterms. Second, if two terms do not share

the same top symbol, we will simplify the terms by assuming the tests that lead to some base case of the inner most recursive call in the first term. By repeating these two procedures, we guarantee that eventually all recursive functions will be removed from the first term. In such a case, one way unification can be used to determine if a definition has been found for any constrained functions remaining. Subterms that contain no constrained functions will be sent back to the prover to determine if they are equal. We do not currently use any time limits for this step but it is possible they may need to be added for more complex theorems.

3.3.2 Discarding Conjectures

Matching creates many subgoals that are easily disproved. We use a simple technique to steer us away from many such goals. Doing so avoids sending the prover down many dead end paths. For formulas where no recursive functions are present, we send the formula through the simplifier. If the simplifier fails to prove the formula, we assume the formula may not be a theorem, and avoid it. For functions with recursive calls, we find any recursive function call in the formula and open it up, creating a number of new formulas with a case split. We then recur on these formulas. In theory, because these functions must terminate, for any false conjecture there exists a finite depth at which this procedure will find a refutation. However, we limit the depth of the search, typically to a maximum of five nested case splits. We have found this technique to be effective for eliminating obviously false conjectures.

3.4 Summary

ACL2 has powerful heuristics which can often prove theorems automatically. However, there are times when several reasonable alternatives exist. Allowing ACL2 to

try more than one alternative and backtrack in the case of failure results in more theorems proved. We presented two such scenarios. First, when an induction scheme contains unmeasured variables, there may be many different viable substitutions for those variables. Second, after induction, it may sometimes be useful to throw away the induction hypothesis and generalize before continuing. Our implementation allows the possibility of extending ACL2 with even more search capabilities. As computers become faster, especially as multi-core processors become more widespread, these search capabilities offer the possibility of taking advantage of this computing power for the purpose of proving theorems. One particularly promising future direction for this work may be to do a parallel implementation of this work.

Chapter 4

Related Work

4.1 Introduction

Theorem provers by W. W. Bledsoe in the late 1960's sometimes contained primitive mechanisms for setting up a single inductive proof over the naturals [3]. Shortly after, Boyer and Moore pioneered the field in the 1970s with the first theorem prover oriented primarily around recursion and induction [19]. Since then, many have contributed to the field. In the following sections, we review some of the most relevant work. Many of these techniques are similar to our work with backtracking in that they look at a goal theorem and do some rewriting or symbolic manipulation to help prove that theorem. However, our technique from Chapter 2 for generating proofs from examples is unique in that it actually looks at entire sequences of rewrites, while most other techniques focus on the induction hypothesis and the induction conclusion. Additionally, instead of trying to guide the rewriter to prove the theorem or suggest lemmas or generalizations, our technique attempts to construct the entire proof directly by generalizing finite cases of the proof. One could say that our technique is focused on the proof object itself while most others are focused on

terms. One consequence of this is that rewrite-centric techniques often find simpler proofs because they can find short-cut lemmas or generalizations. Our technique, on the other hand, has more of a model-based feel to it because the finite cases provide a much richer set of underlying constraints. These constraints sometimes lead us to finding more complex proofs but provide more guidance for finding them. Overall, although many other techniques have the same goal as our system, namely to find generalization or lemmas, few other systems operate in a similar manner. Below we describe a few such systems.

4.2 Rewrite Systems

Most inductive theorem provers are built around a rewrite system. A rewrite system consists of a set of rewrite rules, which are theorems of the form $hypothesis \rightarrow left\text{-}hand\text{-}side = right\text{-}hand\text{-}side$, and a rewrite strategy for applying them. Typically, such systems attempt to prove a theorem by starting from the goal and then repeatedly applying rewrite rules to transform the goal into a sequence of equivalent conjectures. If the goal is eventually rewritten to $TRUE$, then the proof is complete. To apply a rewrite rule to a goal, the hypothesis must be satisfied and the left hand side must match some subterm of the goal. If these conditions are met, the rule is applied by replacing the matched subterm by the right hand side of the rule after instantiating the right hand side using the substitution obtained when matching the left hand side. Along with rewriting, such provers also typically use case-splitting, where the goal is broken up into two or more subgoals; generalization, where the current goal is replaced by a more general one; and, of course, induction.

4.3 Recursion Analysis

The earliest theorem prover oriented toward recursion and induction was the Edinburgh Pure Lisp theorem prover [4]. It evolved into NQTHM [19] and was sometimes called the "Boyer-Moore theorem prover". It dates back to 1971 and ACL2, its direct descendent, is still actively used today. The techniques introduced by Boyer and Moore heavily influenced many subsequent theorem provers. Of these techniques, one of the most influential was *recursion analysis*, which was used to determine the induction scheme used to prove a theorem inductively. The technique is based on the premise that every recursive function definition admits a dual induction scheme, where the base cases of the definition correspond to the base cases of the scheme and the recursive cases give the variable substitutions appropriate for the induction step. Recursion analysis works by first extracting all possible induction schemes using the recursive functions present in the conjecture. Then, it eliminates flawed and redundant schemes and chooses one of the remaining, possibly merging some of the schemes. A scheme is considered flawed if the induction hypothesis that was generated cannot be applied to the resultant induction conclusion at least once, and a scheme is considered redundant if it is identical to another or if another scheme is a repeated application of it.

4.4 Generalization Heuristics

It is well known that some theorems must be generalized in order to be proved inductively. Typical examples include accumulator theorems such as $rev1(x, nil) = rev(x)$ or theorems with overly specialized variables, such as $ap(x, ap(x, x)) = ap(ap(x, x), x)$. One of the first papers to deal extensively with this problem was

written by Raymond Aubin [1]. In it, he describes a theorem prover based on the techniques of Boyer and Moore, but with some additional heuristics for generalization. In the original Boyer-Moore prover, a theorem could be generalized by replacing identical subterms on both sides of an equality with a new variable. This can some times result in non-theorems being generated. Aubin tries to avoid this problem by linking generalization more closely to induction. When considering identical subterms on both sides of an equality for generalization, he limits the search to *primary terms*. Primary terms are terms that occur in controlling positions of recursive functions, such as the first argument to *ap*. His second heuristic is based on comparing the symbolically evaluated induction conclusion to the induction hypothesis. The terms in the induction conclusion that differ are replaced by calls to new functions, and the new induction conclusion is taken as the generalized conjecture. The base cases of these new functions are set by matching the generalized conjecture with the original conjecture, and the recursive cases for these functions are set by matching the induction hypothesis for the generalized conjecture with the simplified induction conclusion of the generalized conjecture. Subsequently, common subexpressions across equality can be generalized to provide a simpler conjecture. One caveat to this technique is that if the difference between the induction hypothesis and the induction conclusion includes an induction variable, definitions in one or the other must be expanded to remove the induction variable from the mismatches.

4.5 Rippling

Rippling is a technique introduced by Bundy et al. [7] for proving inductive theorems that uses annotations to guide the rewriter. It is generally used in the context of constructor based induction, where the induction hypothesis matches the goal

theorem and the induction conclusion adds some type of constructors to the induction variables, although there are ways to adapt it to destructor based induction. It works by first comparing the induction conclusion to the induction hypothesis and annotating the parts of the induction conclusion that differ from the induction hypothesis. These differences are called *wave fronts*, and the goal of rippling is to rewrite the induction conclusion in such a way that the wave fronts propagate to a part of the term where they no longer interfere with the application of the induction hypothesis. This is done by applying *wave rules* to the induction conclusion. Wave rules are rewrite rules annotated to show how they affect wave fronts when applied to annotated terms. They are applied systematically to the induction conclusion using a variety of strategies. The most common strategy is rippling-out, where the wave fronts are propagated to the top of the term. Other strategies include rippling-in, which is used to cancel similar top level terms on both sides of an equality, and rippling-sideways, which is used to move wave fronts into positions where they can match universally quantified variables in the induction hypothesis.

Although rippling itself is generally concerned with proving a theorem after the induction scheme has already been chosen, there are a few techniques that help choose a good induction scheme in the context of rippling. Bundy et al. describe an extension to recursion analysis [8] that does this. The idea is to choose an induction rule that will allow all wave fronts to be rippled at least once. A more powerful technique is called middle-out reasoning [17]. It uses meta-variables to represent the induction scheme chosen and thus is able to begin rippling before the scheme is fixed. During the process of completing the proof, the meta-variables are instantiated and the scheme fixed. This allows rippling to influence the induction scheme without backtracking.

4.6 Divergence Critic

Although rippling provides an effective way to guide a rewrite based theorem prover during induction, there are times when no matter how clever the rewrite strategy is, the theorem cannot be proved without first introducing additional lemmas. Toby Walsh's Divergence Critic [22] addresses this problem by recognizing when a proof is diverging and introducing lemmas to remedy the divergence.

The critic works by first extracting a sequence of related equations from the prover's output. Usually, this is formed from equations taken from a sequence of diverging induction attempts along a single open branch in the proof tree, although any diverging sequence of equations can be used. Successive equations in this sequence are difference matched against each other, which results in annotating each equation with wave fronts that mark the differences between each equation and the previous, similar to the annotations placed on the induction conclusion in rippling. The critic then tries to suggest lemmas that will ripple the wave fronts so that the induction can complete successfully. The two main heuristics for suggesting such lemmas are *cancellation* and *petering out*. Cancellation identifies similar wave fronts on both sides of an equality and proposes lemmas that will ripple the wave fronts to the top of each term where they can be cancelled. Petering-out is used when no matching wave fronts are identified. In this case, lemmas are suggested that will eliminate any wave fronts altogether. Suggested lemmas can also be generalized by replacing identical terms on both sides of an equality with new variables. Other heuristics include fertilization and simplification, which attempt to suggest lemmas which allow the induction hypothesis to be used, or allow recursive definitions to ripple wave fronts further.

4.7 Planning Critic

The idea of a planning critic was introduced by Andrew Ireland in 1992 [11]. It builds upon earlier work by Alan Bundy on proof plans [5]. A proof plan consists of a tactic, which describes at a low level how to apply inference rules to a goal to form a new set of goals, and a method, which describes the preconditions that determine when the tactic should be used along with post conditions describing the overall effect of the tactic. This high level information can be used to control the search space of the theorem prover. Critics extend these ideas further by recognizing when a proof method's preconditions are only partially met, and then suggesting ways to satisfy the remaining preconditions. Such suggestions can include speculating new lemmas or modifying the current goal. Ireland and Bundy use this framework in their 1996 paper [12] to generate lemmas and guide other heuristics such as generalization and induction revision. Many of these techniques are based on rippling in the same way that the Divergence Critic was, but are built with a framework that used explicit plans rather than implicit ones. This gives the critic more information about the current state of the proof.

4.8 Lemma Discovery

Kapur and Subramaniam present another technique for automating induction in their 1996 paper [14]. Although the basic goals of generating lemmas and suggesting generalization are similar to those in the Divergence Critic and the Planning Critic, the means used to accomplish these goals are quite different. Rippling is not used, nor any other annotations. Instead, the driving force is to find the appropriate instantiation for any non-induction variables or generalize the conjecture to include

such variables if none exist. This is done by trying to apply the induction hypothesis to the induction conclusion. Any non-induction variables in the induction hypothesis are replaced by second order variables representing the possible instantiations. If the induction hypothesis can almost be applied to some subterm in the induction conclusion, lemmas are speculated that would allow the application to succeed and a substitution is chosen. Substitutions are constrained by applicable known rewrite rules. Any speculated lemmas are first tested for consistency before a proof is attempted.

4.9 Meta-level inference

In the 80s, B. Silver created a proof planning system called LP [21] that used example proofs to generate strategies for solving mathematical equations. An example could be taken from a text book or any other source and entered into the system for study. The example could contain small skipped steps and the strategies learned from the example could be applied to any new problem. In contrast, our system uses examples generated from the goal theorem to prove that theorem only, and cannot tolerate skipped steps. Additionally, LP focuses on learning general problem solving techniques, while our system focuses on solving a given problem. More recent work in this area has been done by Matthias Fuchs [10], Stephan Schulz [20], and Jamnik et al. [13] and has focused on learning how to choose the best heuristic for resolution style systems.

4.10 Inferring Integer Sequences

While not directly related to proving theorems, the problem of detecting integer sequences has been studied by several, including Colton, Bundy, and Walsh [9], and is closely related to our work. In fact, our process for generating an abstract proof tree is completely oblivious to the fact that the abstraction will be used to complete a proof. The process can be used on any tree structure to form an abstraction. For this reason, our work can be viewed as being more closely related to detecting patterns than proving theorems. The main difference between our abstraction algorithm and detecting integer sequences is that we operate on trees instead of sequences. In this way, our technique can be viewed as a generalization of these techniques. In fact, at the heart of our algorithm we reduce abstracting the tree to the problem of detecting an integer sequence. We currently use a very primitive algorithm for detecting these sequences which only detects linear sequences of the form $ax + b$, but we can imagine using more advanced techniques like those mentioned above.

4.11 Omega proofs

Siani Pearson presented some theoretical results that show that if the omega rule is added to Peano arithmetic, cut elimination holds[18] [2]. The omega rule is an inference rule that essentially states "if each $P(n)$ can be proven in a uniform way (from parameter n), then conclude $\forall n P(n)$." Siani shows that cut elimination in this context can be used to replace the need for generalization and induction. Of course, in order to establish that a given formula can be proven in a uniform way requires a meta-logic, where proofs of provability may involve induction or generalization. Our work in Section 2 is similar in spirit. The abstractions that we form represent

the proof of a given theorem, parameterized by n. In order to check these proofs, however, we avoid using a meta-logic and instead embed the meta-level proof at the object level.

Additionally, she developed a system that used proof instances to form a general proof and used that proof to suggest a cut formula to prove the original theorem. Her pattern recognition algorithms, however, were only able to handle patterns that occurred at the top level, and could not deal with nested recursive functions. As a result, she had to manually supply lemmas in some cases where we do not.

4.12 Proofs with Ellipses

Alan Bundy and Julian Richardson have developed a technique for formalizing proofs involving ellipses [6]. Many people informally use ellipses to represent patterns, for example, 1, 2, ... n. In this paper, Bundy and Richardson formalize such notation and present several proofs using this notation. During this process, they use higher order functions to introduce alternative definitions for many common list processing functions. These definitions are written in such a way that they can be translated into a notation that uses ellipses in a natural way. Many proofs become much simpler with these new definitions. Although overall their technique is different from ours, the definitions that they use and the proofs that they present are in some cases very similar to what our system discovers.

Chapter 5

Conclusions

There are two main contributions from this dissertation. The first is a system that proves theorems by generalizing finite cases. One of the key capabilities of this system is that it is able to find generalizations and lemmas automatically. This allows it to prove theorems that typically take user guidance. The interesting thing about this system is that the technique it uses is very different from how most theorem provers operate. Instead of trying to manipulate the theorem directly, it examines special cases of the theorem in order to learn from them something about the general theorem.

These special cases are used to generate rewrite based proofs for the theorem. We then examine these proofs for recurring patterns, and use the patterns we find to generalize the theorem and suggest lemmas. These lemmas and generalizations form a proof that is checked by ACL2.

The main limitation of this work is that it applies only to theorems that have a linear structure, such as theorems about lists or natural numbers. We attempted to extend our work to include theorems about trees. Although incomplete, we

outlined our attack on this problem in section A. We do feel, however, that it still may be a fruitful avenue for future research, and have outlined a possible attack to this extension. We see no fundamental reason the approach described in the dissertation could not be extended to arbitrary theorems. However, each new class of theorems would require new algorithms for pattern discovery. These algorithms can be quite complex, even for simple patterns such as those involving lists. When different types of patterns are combined, the patterns become even more complex. We feel that any complete approach using this style of proof discovery would not only include a means for recognizing a wide variety of patterns, but also some means to hide unnecessary complexity. One simple way to do this may be to require the user to manually enable and disable specific functions, however a fully automated approach would need to attack this problem head on and find a way to automatically discover what functions needed to be enabled for a given theorem. An alternative way to deal with the problem of algorithmic complexity may be to off-load some of the pattern recognition task on to the user. Since humans are naturally adept at recognizing patterns, some kind of intuitive interface that allows the user to indicate where patterns occur may be another effective approach. This would require some careful thought about the layout of the proofs in which patterns are being discovered, because human pattern recognition is so closely linked to the visual presentation of the data.

The second main contribution is a system that introduces backtracking into ACL2. This allows ACL2 to try more than one induction scheme if one fails. It also allows ACL2 to try more than one technique for generalization. Finally, we implemented a technique for finding substitutions for unmeasured induction variables. All of these fit into the same general framework for backtracking. We believe that

as processing power becomes cheaper the ability to do more search is the key to automating theorem proving.

The main limitation to this work is that there are many different algorithms that could be implemented using this technique but we have only completed a few. We do believe that our approach gives others a good example of a few useful heuristics and a way to implement them.

One important related area of future research is finding a way for the user to interact with the prover when the prover is doing a large amount of search. Traditionally, ACL2 prints out a listing of all the steps that were completed until the prover fails. When doing a large amount of backtracking, this is no longer feasible because it would overwhelm the user. Therefore, finding key subgoals to display to the user, along with some kind of status indicator for a search in process, is one interesting area of research. Along these same lines, rather than restricting the output to completed or failed subgoals, suggested lemmas might be something the user could find helpful.

The key theme throughout this work was to find ways to automate the theorem proving process. We have focused specifically on induction, lemma generation, and generalization. We believe these three areas are key to automating proofs, and hope that our contribution helps move the field of automated theorem proving closer to the ideal of full automation.

Appendix A

Trees

In previous sections, we described an implementation of a system that works for functions that operate on natural numbers and true lists. In this section, we outline an approach for extending this method to work with functions that process trees. This technique is incomplete, but we have outlined what we feel may be a promising approach to this problem.

Previously, we generated functions of the form

$$f(0, x) = B$$
$$f(n, x) = g(f(n-1, x))$$

to represent the evaluation traces that are generated by the theorems we were trying to prove. With theorems that process trees, functions of this form can no longer capture the structure of the resultant traces. For example:

$$cons(cons(caar(x), cdar(x)), cons(cadr(x), cddr(x)))$$

has no linear structure. To remedy this problem, we introduce a new schema. This

```
(defun up (p x)
  (if (endp p)
      x
    (up (cdr p) (if (equal (car p) 0) (car x) (cdr x)))))

(defun next-c (c x p)
  (declare (xargs :measure (acl2-count (up p x))))
  (if (atom (up p x))
      t
    (if (atom c)
        (if c
            t
          (cons nil nil))
      (if (not (and (atom (cdr c))
                    (cdr c)))
          (cons (car c) (next-c (cdr c) x (append p (list 1))))
        (if (not (and (atom (car c))
                      (car c)))
            (cons (next-c (car c) x (append p (list 0))) (cdr c))
          t)))))
```

Figure A.1: The function next-c

schema uses the notion of a cut. Just like the natural number that was used to measure how many steps remained before a given function terminates, a cut will serve the same purpose for tree functions. A cut is a tree that represents how far the expansion of a recursive function over a tree data structure has progressed. When the expansion has completed for a given subtree, we denote the completion with a T. Incomplete subtrees are represented with NIL when they have not been processed at all and with a cons of two cuts representing their partial completion otherwise. We define the next cut c of x at path p using the function $next - c$, shown in Figure A.1.

The function *up* uses a path p to select a subterm of x. The function $next - c$ can be used to generate a sequence of cuts. If we start from the initial cut *nil*, with

path nil and x $cons(cons(1,2), cons(3,4))$, $next - c$ will expand to $cons(nil, nil)$. Repeatedly calling $next - c$ in this fashion will yield the following sequence of cuts:

$$nil$$
$$cons(nil, nil)$$
$$cons(nil, cons(nil, nil))$$
$$cons(nil, cons(nil, t))$$
$$cons(nil, cons(t, t))$$
$$cons(nil, t)$$
$$cons(cons(nil, nil), t)$$
$$cons(cons(nil, t), t)$$
$$cons(cons(t, t), t)$$
$$cons(t, t)$$
$$t$$

Note that this sequence follows the expansion and contraction of a recursive function over tree data. Here, we recur first on the cdr, by expanding a nil to a pair of nils. When an atom is reached, it is converted from a nil to a t to represent its completion. Pairs of ts are contracted back to single ts, until only a single t remains.

Using $next - c$, we can define the induction scheme $tind$ which recurs on $next - c$ for a given c and x at a given path p. The function $tind$ is shown in Figure A.2.

This gives us the infrastructure we need to write proofs about the evaluation of tree functions. As an example of how we can use these definitions, consider the theorem $ifl(x, nil) = fl(x)$. The definitions for ifl and fl are:

```
(defun fl (x)
```

```
(defun tind (c x p)
  (declare (xargs :measure ...))
  (if (or (not (cutp c (up p x)))
          (not (pathp p x))
          (equal c t))
      nil
    (tind (next-c c x p) x p)))
```

Figure A.2: The function tind

```
  (if (atom x)
      (list x)
    (append (fl (car x)) (fl (cdr x)))))
```

```
(defun ifl (x a)
  (if (atom x)
      (cons x a)
    (ifl (car x) (ifl (cdr x) a))))
```

The function fl takes a tree and flattens it, creating a list containing all of the leaves of the tree. The function ifl does the same but uses tail recursion instead of append. We can imagine for a given x, the evaluation of the function ifl could look like this:

$$
\begin{aligned}
&ifl(x, nil)\\
&ifl(car(x), ifl(cdr(x), nil))\\
&ifl(car(x), ifl(cadr(x), ifl(cddr(x), nil)))\\
&ifl(car(x), ifl(cadr(x), cons(cddr(x), nil)))\\
&ifl(car(x), cons(cadr(x), cons(cddr(x), nil)))\\
&cons(car(x), cons(cadr(x), cons(cddr(x), nil)))
\end{aligned}
$$

In order to generalize this evaluation, we need a way to represent the partial evaluations as a function of the current cut. This was done in the linear case using our pattern matching algorithm. Although we believe such an algorithm could also be devised to automatically generate such functions for the tree case, we been unable to create such a thing. Instead, we create the abstraction function by hand and leave the automatic generation of such functions for future work. Below we sketch what such an automatically generated proof might look like. The function we have created for this proof is called $iflpart$. It takes the current cut along with x and returns the evaluation of ifl at path p:

```
(defun fl-l (x)
  (if (endp x)
      nil
    (append (fl (car x))
            (fl-l (cdr x)))))
```

```
(defun tacc (p x)
  (if (or (atom p)
          (atom x))
      nil
    (if (equal (car p) 0)
        (append (tacc (cdr p) (car x)) (list (cdr x)))
      (tacc (cdr p) (cdr x)))))

(defun iflpart (c x p)
  (if (atom c)
      (if c
          (append (fl (up p x)) (fl-l (tacc p x)))
        (ifl (up p x) (fl-l (tacc p x))))
    (if (not (and (atom (cdr c))
                  (cdr c)))
        (ifl (up (append p (list 0)) x)
             (iflpart (cdr c) x (append p (list 1))))
      (iflpart (car c) x (append p (list 0))))))
```

The function *tacc* stands for tree accumulator, and we believe this may be a commonly occurring pattern that may occur in other proofs. The function $fl - l$ takes a list of lists and flattens them into a single list. Using these two functions, we are able to represent the accumulator as a function of the current path. Although we were unable the automate the generation of this function, we believe there is a good chance it is possible. One complication is that the accumulator is represented by the composition of two functions, meaning that we cannot directly generate a

single function to represent this pattern as we did in the linear case. One possible avenue to approach this problem would be to have a large set of common functions that could be applied to various subtrees of x in attempt to find a function that fits the proof exactly.

Using $iflpart$ allows us to rewrite the sequence above as[1]:

$$iflpart(nil, x, nil)$$
$$iflpart((nil.nil), x, nil)$$
$$iflpart((nil.(nil.nil)), x, nil)$$
$$iflpart((nil.(nil.t)), x, nil)$$
$$iflpart((nil.t), x, nil)$$
$$iflpart(t, x, nil)$$

If we define the function $nth - c$ as:

```
(defun nth-c (n c x p)
  (if (zp n)
      c
    (nth-c (1- n) (next-c c x p) x p)))
```

[1]below we use . as infix cons

Then we can rewrite the sequence above as:

$$iflpart(nth - c(0, nil, x, nil), x, nil)$$
$$iflpart(nth - c(1, nil, x, nil), x, nil)$$
$$iflpart(nth - c(2, nil, x, nil), x, nil)$$
$$iflpart(nth - c(3, nil, x, nil), x, nil)$$
$$iflpart(nth - c(5, nil, x, nil), x, nil)$$
$$iflpart(nth - c(7, nil, x, nil), x, nil)$$

Note that we skipped steps 4 and 6 because they are equivalent to steps 3 and 5, respectively. The above reduction indicates why *tind* is a good induction scheme for this formulation.

One thing to note about $iflpart$ is that it does not modify x as it recurs. This is important because sometimes when recurring on the car of a cut, we need to retain informations about the cdr of x. In this case, the accumulator of ifl requires this. To help facilitate representation of the accumulator, we introduced the function tacc. This function returns the list of trees that one would visit using a tail recursive function over trees. We would expect many functions to benefit from this function in representing their accumulators. This function iflpart is broken up into four cases. The first two, where c is an atom, represent when either evaluation is complete or when we have not yet fully explored a subtree of x. These two cases correspond to when the cut is T or NIL, respectively. In the case where the cut is T, we choose to represent the completed evaluation using the function fl. Although in the linear case, all of our abstraction functions followed a schema that did not involve any user generated functions, for the tree case it made more sense to allow such functions. In fact, since it is unlikely that any fixed schema or set of functions could be used to

represent arbitrary partial evaluations, a robust scheme should probably allow the user to specify what functions to use. It is also not surprising that a function used in the statement of the theorem is also useful in describing its partial evaluations. Matching partial evaluations with functions is a rich problem area that we were unable to fully explore. In the case where the cut is NIL, iflpart is merely a call to ifl with the proper accumulator for that path. The next step in this case will be to apply the definition of ifl. For the two remaining cases, the first represents when the inner recursive call of ifl has not yet completed evaluation, and the second case is for the recursive call.

After defining the function iflpart, we can prove a generalization of our goal using the induction scheme *tind*:

```
(defthmd ifl-part3
  (implies
   (and
    (cutp c (up p x))
    (pathp p x))
   (equal (ifl-part c x p) (ifl-part t x p)))
  :hints (("Goal" :induct (tind c x p)  :do-not '(generalize))))
```

This will step through each cut of evaluation until the final cut T. Our final goal is simply an instance of this theorem where p is nil and c is nil:

```
(defthm final
  (equal (ifl x nil)
         (fl x))
  :hints (("Goal" :use (:instance ifl-part3 (p nil) (c nil)) )))
```

Another example of a theorem where this technique could be employed is $rp-nodes(count-nodes(x),x) = sumtree(x)$, where these functions are defined as shown:

```
(defun sumtree (x)
  (if (atom x)
      x
    (+ (sumtree (car x))
       (sumtree (cdr x)))))

(defun rp-cons (x)
  (if (consp x)
      (cond ((consp (cdr x))
             (cons (car x)
                   (rp-cons (cdr x))))
            ((consp (car x))
             (cons (rp-cons (car x))
                   (cdr x)))
            (t (+ (car x) (cdr x))))
    x))

(defun rp-nodes (n x)
  (if (zp n)
      x
    (rp-cons (rp-nodes (1- n) x))))

(defun count-nodes (x)
```

```
(if (consp x)
    (+ 1
       (count-nodes (car x))
       (count-nodes (cdr x)))
  0))
```

The function *sumtree* adds the leaves of a tree and returns their sum. The function $rp - cons$ finds the rightmost cons of two atoms and replaces that cons with the sum of its two components. The function $rp - nodes$ calls $rp - cons$ n times. Let x be a tree of the form '(((a . b) . c) . d). If we evaluate $rp - nodes$ on this tree with $n = 3$, we get the following sequence of terms. Note that in this section we use the notation $(a.b)$ for $cons(a, b)$:

$$rp - nodes(3, (((a.b).c).d))$$
$$rp - nodes(2, (((a + b).c).d))$$
$$rp - nodes(1, (((a + b) + c).d))$$
$$rp - nodes(0, (((a + b) + c) + d))$$
$$(((a + b) + c) + d)$$

We can represent this sequence of steps using the following function:

```
(defun st-part (c x p)
  (if (atom c)
      (if c
          (sumtree (up p x))
        (up p x))
    (cons (st-part (car c) x (append p (list 0)))
          (st-part (cdr c) x (append p (list 1))))))
```

The sequence above now becomes:

$$rp - nodes(3, st - part((((t.t).t).t), x, nil))$$
$$rp - nodes(2, st - part(((t.t).t), x, nil))$$
$$rp - nodes(1, st - part((t.t), x, nil))$$
$$rp - nodes(0, st - part(t, x, nil))$$
$$sumtree(x)$$

Note how the first argument of $st - part$ describes how much of the evaluation of $rp - nodes$ on x remains. This is similar to the function $ifl - part$ in the previous example. One key difference here is that we are interested in a different subset of cuts. The particular cuts we are interested in are the ones that replace a pair of ts with a single t. We refer to such cuts as *contract cuts*, and can write the following functions to compute the nth such cut.

```
(defun contract-cutp (c)
  (if (atom c)
      nil
    (if (and (equal (car c) t)
             (equal (cdr c) t))
        t
      (or (contract-cutp (cdr c))
          (contract-cutp (car c))))))


(defun ncc (n c x p)
  (declare (xargs :measure ...))
  (if (or (not (cutp c (up p x)))
```

```
        (not (pathp p x))
        (equal c t))
    nil
  (if (contract-cutp c)
      (if (zp n)
          c
        (ncc (1- n) (next-c c x p) x p))
    (ncc n (next-c c x p) x p))))
```

Now that we have the above functions, we can rewrite the sequence above as:

$$rp - nodes(3, st - part(ncc(0, nil, x, p), x, nil))$$
$$rp - nodes(2, st - part(ncc(1, nil, x, p), x, nil))$$
$$rp - nodes(1, st - part(ncc(2, nil, x, p), x, nil))$$
$$rp - nodes(0, st - part(ncc(3, nil, x, p), x, nil))$$
$$sumtree(x)$$

This sequence suggests the following lemma, where *maxcc* counts the total number of contract cuts:

$$rp - nodes(n, st - part(ncc(maxcc(x)(-)n, nil, x, p)x, nil)) = sumtree(x)$$

What we are doing here with trees is similar to what we did before with lists. Namely, we are looking at the evaluation of theorems for particular cases and looking for patterns. While in the case for lists, most patterns can be described in terms of a integer being incremented or decremented, with theorems involving trees we have found looking for sequences of cuts makes more sense. In the same way that we were able to generalize the results from looking at a finite list to a theorem about

lists of any size, we can look at a particular tree and generalize this to a theorem about any tree. One of the challenges for such an approach is finding a good basis for recognizing patterns. For example, the concept of cuts, contract cuts, and max cut are all used in the theorem above. We believe such concepts could have broad use across a variety of tree theorems.

If we examine the sequence of rewrites a little more closely, we will realize that we have skipped a few steps. Between each step is a subsequence that involves the function $rp - cons$. For example, between the first two steps, we would find:

$$rp - nodes(2, rp - cons((((a.b).c).d)))$$
$$rp - nodes(2, (rp - cons(((a.b).c)).d))$$
$$rp - nodes(2, ((rp - cons((a.b)).c).d))$$
$$rp - nodes(2, (((a + b).c).d))$$

This sequence suggests the lemma:

$$rp - cons(st - part(ncc(n, c, x, p), x, nil)) = st - part(ncc(n + 1, c, x, p), x, nil)$$

These two lemmas form the basis for a proof of the main theorem using our technique. We have completed a prototype of this proof, available at `http://www.cs.utexas.edu/~jderick/thesis-code.tgz`. We believe by introducing algorithms for finding tree based patterns, these lemmas could be discovered automatically, and by developing a sufficient lemma library, such lemmas could be proved automatically.

Note that the lemma above contains the constant nil in the third argument of $st - part$. In order to prove this lemma, this constant will need to be generalized to p. Although the original goal of this technique was to avoid generalization, the

generalization required in this step is simpler than that required for the original theorem. Namely, it involves replacing identical constants on opposite sides of an equality with a new variable. This type of generalization is already done for non-constants in ACL2, and later in this dissertation we will describe a backtracking prover that allows for generalization of this type of formula.

Bibliography

[1] Raymond Aubin. Some generalization heuristics in proofs by induction. *Actes du Colloque Construction: Amelioration et verification de Programmes.*, 1975.

[2] S. Baker, Andrew Ireland, and Alan Smaill. On the use of the constructive omega-rule within automated deduction. In *Logic Programming and Automated Reasoning*, pages 214–225, 1992.

[3] W. W. Bledsoe. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence*, 2:55–77, 1971.

[4] Robert S. Boyer and J. Strother Moore. Proving theorems about lisp functions. *J. ACM*, 22(1):129–144, 1975.

[5] Alan Bundy. The use of explicit plans to guide inductive proofs. In *Conference on Automated Deduction*, pages 111–120, 1988.

[6] Alan Bundy and Julian Richardson. Proofs about lists using ellipsis. In *Logic Programming and Automated Reasoning*, pages 1–12, 1999.

[7] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[8] Alan Bundy, Frank van Harmelen, Jane Hesketh, Alan Small, and Andrew Stevens. A rational reconstruction and extension of recursion analysis. In *IJCAI*, pages 359–365, 1989.

[9] Simon Colton, Alan Bundy, and Toby Walsh. Automatic identification of mathematical concepts. In *Proc. 17th International Conf. on Machine Learning*, pages 183–190. Morgan Kaufmann, San Francisco, CA, 2000.

[10] Matthias Fuchs. A feature-based learning method for theorem proving. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 457–462, Menlo Park, July 26–30 1998. AAAI Press.

[11] Andrew Ireland. The use of planning critics in mechanizing inductive proofs. *Lecture Notes in Artificial Intelligence*, 624:178–189, 1992.

[12] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[13] Mateja Jamnik, Manfred Kerber, Martin Pollet, and Christoph Benzmueller. Automatic learning of proof methods in proof planning. Technical Report CSRP-02-05, University of Birmingham, School of Computer Science, June 2002.

[14] D. Kapur and M. Subramaniam. Lemma discovery in automating induction. *Lecture Notes in Computer Science*, 1104:538–??, 1996.

[15] M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[16] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *Software Engineering*, 23(4):203–213, 1997.

[17] Ina Kraan, David A. Basin, and Alan Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1-2):113–145, 1996.

[18] Siani Pearson. Linearisation of omega proofs: A new method of generalisation within automated deduction. Technical Report HPL-95-93, HP Laboratories Bristol, August 1995.

[19] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1979.

[20] Stephan Schulz. Learning search control knowledge for equational theorem proving. *Lecture Notes in Computer Science*, 2174:320–, 2001.

[21] B. Silver. *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence*. North Holland, 1985.

[22] Toby Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

Vita

John was born in Rockford, IL., and went to high school in Albuquerque, NM. He came to UT for college, and graduated with a degree in Electrical Engineering. After working for a while, he returned to graduate school to pursue his PhD in Computer Science. He is married and has a daughter.

Permanent Address: 907A Possum Trot St
Austin, TX 78703

This dissertation was typeset with LaTeX 2_ε[2] by the author.

[2] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.